CB-80™

LANGUAGE MANUAL

DIGITAL RESEARCH™

CB-80™ Language

Version 1.3 Update

Release Notes

Compiled April 1982

Copyright © 1982

Digital Research
P.O. Box 579
160 Central
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

All Rights Reserved

CB-80™ Language

Version 1.3

Enhancements

CP/M is a registered trademark of Digital Research
CB-80, LK-80, and MP/M II are trademarks of Digital Research
Copyright © 1982 by Digital Research. Inc.

The following are enhancements to CB-80™ Version 1.3.

1) Fatal compiler errors give the source line number that caused the error. This is almost always due to a syntax error. Please submit the statement on a Software Performance Report (SPR) to Digital Research.

2) A new compiler error message (number 6) was added to indicate that statements following an END were encountered.

3) A new function has been added to CB-80. The ERRX function returns the sixteen-bit MP/M II™ extended error code. A new execution error EX has also been added, and indicates an extended error has occurred. CB-80 now traps all errors under MP/M II.

4) The linker LK-80™ has significant enhancements that the LK-80 Version 1.3 supplement describes.

5) The INITIALIZE statement is enhanced to support resetting specific drives under MP/M II.

```
INITIALIZE <exp>
```

initializes the drives selected by the integer expression. Bit 0 is drive A; bit 1 is drive B, etc.

```
INITIALIZE 4
```

resets drive C. Under CP/M® Version 2.2 the expression is ignored, and all drives are reset. Under MP/M II, if the expression is 0, all drives are also reset.

All Information Presented Here is Proprietary to Digital Research

Invoke LIB by typing:

```
LIB filename=filename1,...,filenameN
```

This command creates a library called filename.REL from the files filename1.REL,...,filenameN.REL. If you omit filetypes, REL is assumed.

A filename can be followed by a group of module names enclosed in parentheses. Only the modules enclosed in parentheses are included in the LIB function being performed. If you omit the parentheses, all modules in the file are included, for example:

```
LIB TEST=A(A1,A2),B,C(C1-C4,C6)
```

This command creates a file TEST.REL that consists of modules A1 and A2 from A.REL, all the modules from B.REL, the modules C1 through C4, and C6 from C.REL.

You can include any of several optional switches in the command line for LIB. Enclose these switches in square brackets and place them after the first filename in the LIB command. The switches are:

I	creates an indexed library file (IRL)
M	prints module names
P	prints module names and public symbols

For example,

```
LIB TEST=A,B,C
```

creates a file TEST.REL that consists of A.REL, B.REL, and C.REL.

```
LIB TEST=TEST,D
```

appends D.REL to the end of TEST.REL.

```
LIB TEST[I]
```

creates an indexed library TEST.IRL from TEST.REL.

```
LIB TEST[I]=A,B,C,D
```

performs the same function as the preceding LIB examples, except no TEST.REL file is created; an indexed library TEST.IRL is created.

```
LIB TEST[P]
```

lists all the module names and PUBLICS in TEST.REL.

A list of error messages generated by LIB is located at the end of this addendum.

All Information Presented Here is Proprietary to Digital Research

LK-80™ ENHANCEMENTS

Enhancements to Section 12 of the CB-80™ Language Reference Manual

CB-80 and LK-80 are trademarks of Digital Research
Copyright © 1982 by Digital Research, Inc.

Version 1.3 of LK-80™ provides many enhancements to LK-80 Version 1.2. LK-80 Version 1.3 has more space for symbols and allows more external-plus-offset references, among other significant enhancements. This addendum provides a full explanation and examples of the new features, with a final note on external-plus-offsets.

LK-80 Version 1.3 includes the following new features:

- support for user library files
- linking one overlay separately from the root
- listing the starting address of all modules to the console as they are linked
- two new command-line toggles, M and S

User Libraries

LK-80 version 1.3 supports user indexed library files. A library file consists of one or more relocatable (REL) modules. A library file differs from an indexed library file. An indexed library file consists of a library preceded by an index of all the public symbols that occur in the constituent modules. The index allows LK-80 to quickly determine which, if any, of the modules in the library are required to resolve external symbols. An indexed library file has the filetype IRL (indexed relative).

An example of an indexed library file is CB80.IRL. (This file is included on the CB-80™ distribution diskette.) After reading all the REL files, LK-80 always reads the index of CB80.IRL to search for any remaining unresolved symbols. If unresolved symbols are found, LK-80 links only those modules in which these symbols are defined. For example, if a module in one of your overlays requires the square root subroutine, LK-80 notices that the symbol for this routine (?rsqr) is in the index of CB80.IRL, and that this symbol is not defined in any the programs you are linking. This causes LK-80 to link the module in which this symbol is defined (CBTRAN). LK-80 does not link any of the other modules in CB80.IRL into your program unless they contain symbols required by any of your modules.

The librarian program, LIB.COM, creates library files. The CB-80 Version 1.3 distribution diskette includes the LIB.COM program. LIB is a versatile utility that handles libraries. LIB.COM can:

- concatenate a group of REL files into a library
- create an indexed library
- select modules from a library
- print module names and public symbols from a library.

All Information Presented Here is Proprietary to Digital Research

satisfy externals from SWEET and DRY. C3.IRL and C4.IRL are specified within the parentheses that delimit overlay 1 modules. This does not indicate a special relationship between these indexed libraries and this overlay. The following command line has precisely the same effect:

```
LK80 C3.IRL,C4.IRL,SWEET,C2.IRL(DRY,C1.IRL)
```

If you specify more than 10 indexed library files in the command line, LK-80 aborts and displays the following message:

```
TOO MANY LIBRARY FILES
```

If this occurs, simply make new, larger libraries from the IRL files you are linking.

You cannot extract more than one hundred modules from all indexed libraries (including CB80.IRL) during one link. If you try to extract more than one hundred modules, LK-80 aborts with the following message:

```
TOO MANY LIBRARY MODULES
```

If this occurs, combine some of the constituent REL files in the libraries into larger files, so that the libraries contain fewer modules.

A library file index must not exceed 16k. An index of 16k contains over 1600 symbols, so you can easily maintain this limit. Also, because LK-80 keeps the entire index in memory while the library is being processed, it competes with the symbol table for space. Consequently, if the index is larger than 16k, LK-80 aborts with the following message:

```
INDEX TOO BIG
```

In LK-80 Version 1.3, the symbol table grows upward in available memory, while any library file indices read in downward from the top of available memory. The message "symbol table space remaining," which is the last line of LK-80 output, reports the minimum size of the gap between the top of the symbol table and the bottom of the index. If the "symbol table overflow" or "not enough memory" messages appear, a collision between these two boundaries occurred. The following list describes three methods of dealing with this situation.

1) If you are linking several indexed library files, specify them in descending order by size, so the largest is linked first while the symbol table is smaller.

2) Break down large library files to reduce index size.

All Information Presented Here is Proprietary to Digital Research

Libraries and Linking

The following command creates a file named BIGREL.REL by concatenating the files SMALL1.REL, SMALL2.REL, and SMALL3.REL. In this example, no index is included in the output file.

```
LIB BIGREL=SMALL1,SMALL2,SMALL3
```

The three input files are not changed in the process. You can use this method to unite programs that are always required in combination. For example, SMALL2 and SMALL3 might contain public functions that are always needed by SMALL1. Thus, combining these files simplifies the LK-80 command lines that refer to them.

Perhaps these three files are not interdependent, and yet they all support the same types of applications. Many different programs might require one or two of them, but rarely all three. In this case, it is convenient to create an indexed library file that contains all three modules. Then, you can include the library filename in the LK-80 command line whenever the application program might require one or more of them. Thus, you do not have to think about which module is needed in a particular link. LK-80 figures this out for you, and includes only those needed. This way, the COM file is no larger than necessary.

All user-specified indexed library modules are contained in the root. A reference to an indexed library can occur anywhere in a command line where a REL filename is legal. However, any modules extracted from these libraries are placed in the COM file, above the CB-80 run-time routines and below the overlay code area. This reduces both the disk space required to store overlay files and the time required to load overlays during chaining.

In the following example, the command creates a root CLARET.COM and overlays BURGUNDY.OVL and CHABLIS.OVL. The indexed library file MYLIB.IRL satisfies externals before the default library (CB80.IRL) is read. The root CLARET.COM includes any modules extracted from MYLIB.IRL.

```
LK80 CLARET,MYLIB.IRL(BURGUNDY)(CHABLIS)
```

You can specify up to 10 indexed library files in the LK-80 command line. They are processed in the order of their occurrence, so the first module that contains a definition of a particular external is the only such module linked. This also applies to symbols defined in CB80.IRL. Therefore, you can define your own version of various run-time routines.

The following command generates the root SWEET.COM and the overlay DRY.OVL from the object files SWEET.REL and DRY.REL.

```
LK80 SWEET,C1.IRL,C2.IRL(C3.IRL,C4.IRL,DRY)
```

The indexed library files C1, C2, C3, and C4 are all processed to

All Information Presented Here is Proprietary to Digital Research

library names in the command line.

If present, the LNK filename must precede any other filename in the command line. LK-80 ignores any that precede LNK. All filenames after the LNK filename are assumed to constitute one overlay. Filenames within parentheses are ignored, because only one overlay at a time can be short-linked. If two or more overlays have been changed, run LK-80 once for each overlay in short mode.

The following command creates HAMPSTER.OVL by linking HAMPSTER.REL and GOLDFISH.REL using data supplied by the PET.LNK file.

```
LK80 DOG,CAT,PET.LNK,HAMPSTER,GOLDFISH(COW)
```

PET.LNK contains the environment data from the link that created the root PET.COM. LK-80 ignores the filenames DOG and CAT because they precede the LNK filename. The file COW is also ignored because it is in parentheses.

The following command does not create a new root.

```
LK80 PET.LNK,PET
```

It creates a file PET.OVL that functions as an overlay, but that does not replace PET.COM as a new root. If the root is changed, a full link of the program is required. Thus, if you are developing a large program with many overlays and want to use short-linking to maximum advantage, you should make the root as simple as possible, so you do not have to change it often.

Note: a symbol file is not generated during a short link.

Load Mapping with the M Toggle

The new M toggle causes LK-80 to output all module names followed by their absolute starting addresses. Library modules are listed first, followed by the overlay modules.

You can use the load map with the addresses provided in the CB-80 listing to aid in debugging. When a CB-80 run-time error occurs, the run-time system prints out the error code followed by the address where it was detected. This address, minus the load address of the module executing when the error occurred, gives the offset of the error address. This offset can then be compared with the CB-80 interlisting (I toggle) to find the source line that probably caused the error.

If you specify multiple toggles within one pair of square brackets, separate them by commas.

All Information Presented Here is Proprietary to Digital Research

3) If you are linking many overlays and a certain REL file is specified as part of several overlays, make the REL file into an indexed library file. This makes the REL file part of the root, which considerably reduces the memory required for the symbol table.

Relinking One Overlay with the S Toggle

LK-80 Version 1.3 can relink one overlay independent of other overlays with which it was created. To do this, LK-80 saves certain information each time it links the root used with this overlay. The new S toggle commands LK-80 to save this information in a file with the same name as the root, but with a filetype of LNK. In subsequent invocations of LK-80, the appearance of the filetype LNK in the command line causes LK-80 to read this file, and to use the data it contains to link the overlay specified in the remainder of the command.

The following command creates the root BREAKFST.COM, and overlays EGGS.OVL, COFFEE.OVL, and CEREAL.OVL. The S toggle saves information for future links in a file named BREAKFST.LNK.

```
LK80 BREAKFST,BUTTER,JAM(EGGS,BACON)(COFFEE,CREAM,SUGAR)(CEREAL)[S]
```

Suppose that at run time, you detect an error in CEREAL. Correct the error and then recompile the source to form a new CEREAL.REL. You can then link this file into the BREAKFST program system without repeating the full link specified in the example.

The following command tells LK-80 to create a file CEREAL.OVL from the input file CEREAL.REL using the information in BREAKFST.LNK. LK-80 completes this command in much less time than the previous command.

```
LK80 BREAKFST.LNK,CEREAL
```

This short-cut method of relinking one overlay works if the following conditions are satisfied:

- The new overlay must not require code, common, or data segment sizes larger than the respective segment sizes from the original full link. If one of these segments exceeds the corresponding segment size in the LNK file, LK-80 aborts with the message

  xxxx size exceeded. Short link aborted.

  where xxxx is Code, Common, or Data, depending on which segment is too large. If this occurs, you must reduce that segment of the overlay, or fully relink the overlay.

- The new overlay must not reference library modules that are not included in the root. For this reason, LK-80 ignores any

All Information Presented Here is Proprietary to Digital Research

```
extrn   lookup,var,newlock

call    lookup + 3      ;Call to 3 bytes past start of "lookup"
lda     var + 1         ;Get high order byte of 16-bit variable
jmp     newloc + 12     ;Branch to location 12 bytes past "newloc"
```

Each of the last three lines generate external-plus-offset records.

If the "too many external-plus-offsets" message appears, you must create symbols for some of these values. For example, in the module that defines "lookup", give the location "lookup + 3" a public name (perhaps "lookup3") and add this name to the external symbols list in the preceding program.

Lib Error Messages

The following table presents LIB error messages and their meanings.

Table 1. LIB Error Messages

Message	Meaning
CANNOT CLOSE	The output file cannot be closed. The diskette might be write-protected.
DIRECTORY FULL	There is no directory space for the output file.
DISK READ ERROR	A file cannot be read.
DISK WRITE ERROR	A file cannot be written, probably due to a full diskette.
FILE NAME ERROR	The form of a source filename is invalid.
NO FILE	The indicated file cannot be found.
NO MODULE	The indicated module cannot be found.
SYNTAX ERROR	The command line to invoke LIB was not correctly formed.

All Information Presented Here is Proprietary to Digital Research

Summary of New Features

The following command produces the root COLORS.COM, and the overlays GREEN.OVL, RED.OVL, and BLUE.OVL from the files COLORS.REL, GREEN.REL, RED.REL, and BLUE.REL.

```
LK80 COLORS,PASTELS.IRL,HUES.IRL(GREEN)(RED)(BLUE)[S,Q,M]
```

The library files PASTELS.IRL and HUES.IRL resolve unsatisfied external references from the REL files before CB80.IRL is used for the same purpose. Any modules in PASTELS and HUES that are required to complete the program are included in the root. This is true even if these library names appear within parentheses.

LK-80 produces the link environment file COLORS.LNK for future use in short-linking changed versions of the preceding overlays or new overlays. A memory map that shows the starting addresses of all modules, including any modules required from PASTELS and HUES, is generated on the console as the link proceeds. Finally, the symbol file COLORS.SYM includes all symbols that start with ?.

The following command creates an overlay VIOLET.OVL from the file VIOLET.REL, using the link environment file generated in the previous command. LK-80 ignores the library name SHADES.IRL because short-linking does not allow new libraries to be referenced.

```
LK80 COLORS.LNK,VIOLET,SHADES.IRL
```

Note on External-Plus-Offset References

Previous versions of LK-80 generated the message "LK80 failure 2" when the table that saves external-plus-offset references overflowed. In Version 1.3, the message is:

```
TOO MANY EXTERNAL-PLUS-OFFSETS
```

Furthermore, the table size in Version 1.3 is significantly larger, so this error should occur less frequently. The next few paragraphs discuss the meaning of external-plus-offset references.

References to offsets from external symbols usually occur in assembly language programs. CB-80 does not generate them, and the same is probably true of most other compilers. Whenever an assembly language instruction has an operand expression that consists of an external symbol and an added or subtracted constant, the assembler emits an external-plus-offset record into the REL file. When LK-80 determines the value of this symbol, it then adds the offset to give the correct run-time value of the operand. The following examples come from a fragment of an RMAC program.

All Information Presented Here is Proprietary to Digital Research

CB-80™ Language

Version 1.3

Corrections

CP/M is a registered trademark of Digital Research
CB-80, MP/M, and MP/M II are trademarks of Digital Research
Copyright © 1982 by Digital Research, Inc.

The following errors in CB-80™ are corrected in Version 1.3 of CB-80.

1) A PRINT USING statement terminated by two semicolons caused a fatal compiler error.

2) Under MP/M II™, the ATTACH function does not operate correctly.

3) After chaining, the system failed to recognize when it was operating under MP/M™.

4) You could not always chain from another user area to a program with the SYS and RO attributes located in user 0.

5) If a label was not referenced in a program, the value for the label in the SYM file was incorrect.

6) The toggle S caused multiple definition errors when linked with public functions.

7) A user in a user number other than 0 could not open a system file in user 0.

8) If a file print statement was improperly terminated by a semicolon, a fatal compiler error resulted.

9) The INKEY function now operates properly with CP/M® 2.2.

10) If a labeled remark was followed by a declaration statement, an incorrect error message was generated.

11) Some string comparisons did not operate properly.

12) Improper code was generated for some exponentiation operations.

13) The compiler would hang if the last record of the source ended on a 128-byte boundary, and the last character in the file was a tab character.

14) If a program started with an alphanumeric label, a spurious compiler error occurred.

15) VARPTR did not always return the correct value.

16) Exponentiation with large negative exponents did not return the correct value.

All Information Presented Here is Proprietary to Digital Research

CB-80 T.M. 1.2 Enhancements 12/28/81

Copyright © 1981 by Digital Research, Inc., Pacific Grove, CA 93950

1. The memory allocation routines were improved to reduce fragmentation of memory.

2. The ATTACH function was not properly documented in the CB-80 1.1 Enhancement notes.

   ATTACH (printer%)

   returns an integer vavle -1 if the logical printer (printer%) is attached otherwise a 0 is returned.

```
IF NOT ATTACH (printer%) THEN
CALL NO. PRINT
```

3. The LOCK and UNLOCK functions return a 0 if the operation was successful. Otherwise the error code returned by MP/M is returned.

4. LK80 supports a maximum of 60 overlays. The total number of modules linked may not exceed 60, regardless of the number of overlays.

All Information Presented Here is Proprietary to Digital Research.

Known Problems With CB-80™ V1.2

Copyright ©1981 by Digital Research, Inc., Pacific Grove, CA 93950

1. The INKEY function will not operate properly with the CONSTAT% function due to a known bug in CP/M 2.2. This will be corrected in a future release of CP/M 2. INKEY operates properly with MP/M version 2.

2. A multiple line function which returns a string may not be referenced twice within the same expression.

```
DEF    FNA$
       FNA$=A$+B$
FEND
PRINT FNA$+FNA$
```

This is a limitation in the implementation of CB-80 Version 1.

All Information Presented Here is Proprietary to Digital Research.

CB-80™ 1.1 Enhancements 11/20/81

Copyright© 1981 by Digital Research, Inc., Pacific Grove, Ca 93950

Additions and enhancements in version 1.1 of CB-80:

1. When LK-80™ cannot open a file, the name of the file is printed as part of the error message.

2. The R toggle was added to allow the "REL" file to be placed on a drive other than the one containing the source file.

   EXAMPLE: CB80 B:TEST[R(C)]

   This command line compiles TEST.BAS from Drive B placing the TEST.REL file on Drive C.

3. Support for MP/M II™ has been enhanced. Three file attributes have been added to the OPEN and CREATE statements. These attributes correspond to the same used by MP/M II. A file can now be opened as either LOCKED, UNLOCKED, or READONLY.

   ```
   EXAMPLE: OPEN "TEST" as 2 READONLY
            CREATE A$ RECL 100 AS I% UNLOCKED
            OPEN "ACCOUNTS" AS 4 LOCKED
   ```

4. An ATTACH function was added to allow a printer to be attached. The syntax for this function is ATTACH <numeric expression>. The example below attaches printer number 3.

   EXAMPLE:ATTACH 3

5. A DETACH statement was added to detach the currently attached printer.

   EXAMPLE: DETACH

6. LK-80 an accept the command line from a file by preceding the filename with a less than sign (<).

   EXAMPLE: LK80 < LINK.SRC

   LINK.SRC is the name of the file containing the command line for LK-80. A blank character must precede and follow

All Information Presented Here is Proprietary to Digital Research

the less than sign.

7. When the compiler lists to a printer, a form-feed is the first character output.

CB-80, LK-80, and MP/M II are trademarks of Digital Research.

All Information Presented Here is Proprietary to Digital Research

TABLE OF CONTENTS

1. INTRODUCTION TO CB8Ø **3**
1.1. CB8Ø Program Primitives 2
1.1.1. CB8Ø Character Set 2
1.1.2. Identifiers and Reserved Words 3
1.1.3. Constants 5
1.1.4. Remarks 8
1.2. Notation 9

2. CB8Ø PROGRAM STRUCTURE **11**
2.1. CB8Ø Programs 11
2.2. Compiler Directives 15
2.2.1. Listing Control Directives 15
2.2.2. %INCLUDE Directive 16

3. DATA TYPES AND DECLARATIONS **18**
3.1. Numeric Data 18
3.2. String Data 19
3.3. Label Data 19
3.4. Data Structures 2Ø
3.5. Declarations 21
3.6. Default Declarations 23
3.7. DATA Statements 23
3.7.1. Identifier Usage 24

4. USER DEFINED FUNCTIONS **26**
4.1. Introduction to Functions 26
4.2. Single Line Functions 27
4.3. Multiple Line Functions 28
4.4. Scope of Variables 3Ø

5. EXPRESSIONS AND ASSIGNMENTS **32**
5.1. Operands 32
5.2. Operators 34
5.2.1. Logical Operators 35
5.2.2. Relational Operators 37
5.2.3. Arithmetic Operators 38
5.2.4. Assignment Statements 41
5.2.5. Evaluation of Expressions 42
5.3. Mixed Mode Expressions 43

6. PREDEFINED FUNCTIONS **44**
6.1. Numeric Functions 44
6.1.1. ABS 44
6.1.2. ATN 44
6.1.3. COS 44
6.1.4. EXP 45
6.1.5. FLOAT 45
6.1.6. INT and INT% 45
6.1.7. LOG 45
6.1.8. MOD 45

6.1.9. SGN 46
6.1.10. SIN 46
6.1.11. SQR 46
6.1.12. TAN 46
6.2. String Functions 47
6.2.1. ASC 47
6.2.2. CHR$ 47
6.2.3. LEFT$ 47
6.2.4. LEN 47
6.2.5. MATCH 48
6.2.6. MID$ 49
6.2.7. RIGHT$ 49
6.2.8. STR$ 49
6.2.9. UCASE$ 50
6.2.10. VAL 50
6.3. Miscellaneous Functions 50
6.3.1. COMMAND$ 50
6.3.2. ERR 51
6.3.3. ERRL 51
6.3.4. FRE 51
6.3.5. MFRE 51
6.3.6. SADD 52
6.3.7. VARPTR 52

7. FLOW OF CONTROL STATEMENTS 53
7.1. GOTO Statements 53
7.2. IF Statements 54
7.3. FOR Loops 56
7.4. WHILE Loops 59
7.5. GOSUB Statements 60
7.6. CALL Statements 61
7.7. RETURN Statements 62
7.8. ON Statements 63
7.9. ON ERROR Statements 65
7.10. STOP Statements 66

8. INPUT/OUTPUT PROCESSING STATEMENTS 67
8.1. INPUT Statements 67
8.2. CONSOLE and LPRINTER Statements 70
8.3. PRINT Statements 71
8.4. POKE Statements 74
8.5. OUT Statements 74
8.6. READ Statements 75
8.7. RESTORE Statements 76
8.8. RANDOMIZE Statements 76
8.9. Input/Output Predefined Functions 77
8.9.1. CONSTAT% 77
8.9.2. CONCHAR% 77
8.9.3. INKEY 78
8.9.4. INP 78
8.9.5. PEEK 78
8.9.6. POS 79
8.9.7. RND 79
8.9.8. TAB 79

9. FILE PROCESSING STATEMENTS..... 80
9.1. File Description..... 80
9.2. OPEN and CREATE Statements..... 81
9.3. File Accessing Methods..... 83
9.3.1. Reading Files..... 84
9.3.2. Writing to Files..... 87
9.4. Terminating Access To Files..... 89
9.5. File Exception Processing..... 90
9.6. File Predefined Functions..... 91
9.6.1. GET..... 91
9.6.2. LOCK..... 91
9.6.3. RENAME..... 92
9.6.4. SIZE..... 92
9.6.5. UNLOCK..... 93

10. FORMATTED OUTPUT..... 94
10.1. Using Strings..... 94
10.2. Numeric Fields..... 95
10.3. String Fields..... 99
10.4. Escape Characters..... 102
10.5. Print Using to Files..... 103

11. PROGRAM MODULES..... 104
11.1. Public and External Functions..... 104
11.2. Linkage With Assembly Language Routines..... 105
11.3. Chaining to Another Program..... 106

12. COMPILER OPERATION..... 108
12.1. Compiling a Program..... 108
12.2. Compiler Toggles..... 109

Appendices

A. CB80 RESERVED WORDS..... 112

B. COLLECTED SYNTAX DIAGRAMS..... 113

C. COMPILER ERROR MESSAGES..... 129

D. EXECUTION ERROR MESSAGES..... 137

E. IMPLEMENTATION DEPENDENT VALUES..... 140

F. SUBJECT INDEX..... 142

1. INTRODUCTION TO CB80

CB80tm is a high level language designed to implement commercial applications on 8080 and Z80tm based microcomputer systems. CB80 maintains compatibility with Compiler Systems' language CBASICtm while providing significant performance advantages. In addition CB80 provides the support required for operation in a multi-user environment. CB80 consists of the CB80 compiler, the CB80 library and a link editor, LK80tm.

The CB80 language has been designed to implement large business systems. A great deal of effort has gone into ensuring that very large programs can be compiled and that performance does not degrade when applications become large.

The CB80 compiler translates CB80 statements into a module consisting of relocatable machine instructions. A number of separately compiled modules may be combined into a single relocatable module with LK80. A relocatable module linked with the CB80 library by LK80 produces an executable program. Overlays may be generated to allow one program to chain to another.

CB80 and associated programs are distributed by Compiler Systems Inc. or by dealers licensed by Compiler Systems to distribute CB80. A diskette containing an authorized copy of CB80 will have a label similar to the one shown below.

COMPILER SYSTEMS, INC.

Version _____

Serial # _____

CSI Dealer _____

CB80™

END USER DISKETTE

NOTICE OF RESTRICTIONS:
All software on this diskette is copyrighted and may be used and copied only under the terms of the Compiler Systems, Inc. End User License Agreement. This diskette is serialized and may be used only by the registered user, and may not be resold or transferred without the consent of Compiler Systems, Inc., P.O. Box 145, Sierra Madre, California.
CB80 is a trademark of Compiler Systems, Inc.
Copyright © Compiler Systems, Inc. 1981

A copy of Compiler Systems' CB80 Licensing Guide is provided with each copy of CB80. If you do not have a Licensing Guide or if your disk does not have an end user label, please contact Compiler Systems at 213-355-4211.

CSI is very interested in your comments on our documentation and programs. Included with your distribution disk are some problem report forms. Please use them to help us provide you with a better product.

CB8Ø and all programs distributed with CB8Ø are protected by public laws 94-553 and 96-517. The programs and all the documentation are copyrighted by Compiler Systems. An infringement of our copyright for commercial advantage or financial gain subjects an individual to federal criminal prosecution. It is a corporate objective of Compiler Systems to prosecute infringements of our copyrights to the full extent of the law.

1.1. CB8Ø Program Primitives

A CB8Ø program is a text file consisting of ASCII characters. In this manual the program will be called the source program or source code. A physical line of source code is terminated by the end of line character which is normally a carriage return followed by a linefeed.

Groups of characters form one of the following program primitives: identifiers, reserved words, constants, special characters, or remarks. Any number of blanks may be inserted between program primitives. Except in a string constant, a consecutive group of blanks is treated as one blank. For example:

```
PRINT X
```

and

```
PRINT                    X
```

are treated as the same two primitives.

1.1.1. CB8Ø Character Set

Any ASCII character may appear in a CB8Ø program. The CB8Ø language uses the alphanumeric characters and the following special characters:

```
! " # $ % & ( ) = - ^ \ * : + ; ? / > . < ,
```

Tab characters may be used in source programs; they are treated as blank characters. Listings expand tabs to the next column that is a multiple of eight.

Except in string constants, lower-case alphabetic characters are converted to the corresponding upper-case alphabetic characters.

The following primitives are considered the same:

PRINT

and

print

The backslash character (\) has special meaning in CB8Ø. Unless it is contained in a string constant, explained in section 1.1.3, the backslash signifies that the next line is a continuation of the current line. This allows a line oriented language like Basic to have statements extend over many physical lines. Any characters following the backslash on the same physical line are ignored. The use of continuation characters is explained in more detail in chapter two.

1.1.2. Identifiers and Reserved Words

An identifier is a string of alphanumeric characters and decimal points. Some identifiers may end with a percent sign (%) or a dollar sign ($). The first character must be alphabetic or a question mark (?). CB8Ø permits a question mark (?) as the first letter of an identifier for access to library routines in the CB8Ø runtime library.

There are also some identifiers with special meaning to the compiler that start with a percent sign (%). These special identifiers are called compiler directives. They are explained in chapter two. Appendix A contains a list of all compiler directives.

Lower-case letters in an identifier are always converted to the corresponding upper-case letters.

A special type of identifiers are reserved words. These are identifiers that have specific meaning in the CB8Ø language. Appendix A contains a list of CB8Ø reserved words. In the remainder of this manual the term identifier will be used to refer to identifiers other than reserved words or compiler directives.

Identifiers are used to represent program elements, defined by the programmer, such as variables, function names and labels. Subsequent chapters will explain the use of these elements. In general the same identifier may not be used for two different elements. This will be explained in greater detail in later chapters.

Identifiers may be of any length; however a specific implementation may limit the number of characters that are significant. In no case will CB8Ø set this limit to less than 31 characters. See appendix E for current implementation limits.

Names of functions which are public or external may be truncated to less than 31 characters due to limitations imposed by linkage editors.

If an identifier is truncated due to an implementation length restriction, a terminating dollar sign or percent sign will not be retained. This could alter the expected operation of the program.

The following list shows valid identifiers:

AMOUNT	FN.ANGLE
PAYMENT.DUE.DATE	ORDER.QTY
INDEX%	I%
DaTe$	ACCOUNT.1Ø1
income.source.code	A1.B2.C3.
?GETS	I

Long identifiers are provided so that programmers may choose names that have meaning. This makes programs easier to develop and maintain. The amount of space CB8Ø requires during compilation of a program is related to the length of identifier names. Thus a large number of extremely long variable names could limit the size of the program that may be compiled. This should not be a practical limit. In no case does the size of identifiers affect the size of the executable code produced by CB8Ø.

The following identifiers are <u>invalid</u>:

A$%	$ is only allowed at the end
SIN	SIN is a reserved word (see appendix A)
7IJK	must start with a letter (I7JK is OK)
A?B	question marks may only appear at the beginning of an identifier
$	must start with a letter or ?
.A.B	decimal points may not start an identifier
A B C	spaces may not appear in identifiers

Decimal points are imbedded in an identifier to enhance readability. Any number of decimal points may be used and they may appear at the end of the identifier. For example:

```
MASTER.ACCOUNT.NUMBER.

FILE.NUMBER%
```

The decimal point is part of an identifier and must be present in all references to that identifier. In other words the identifiers:

```
INV.NO
```

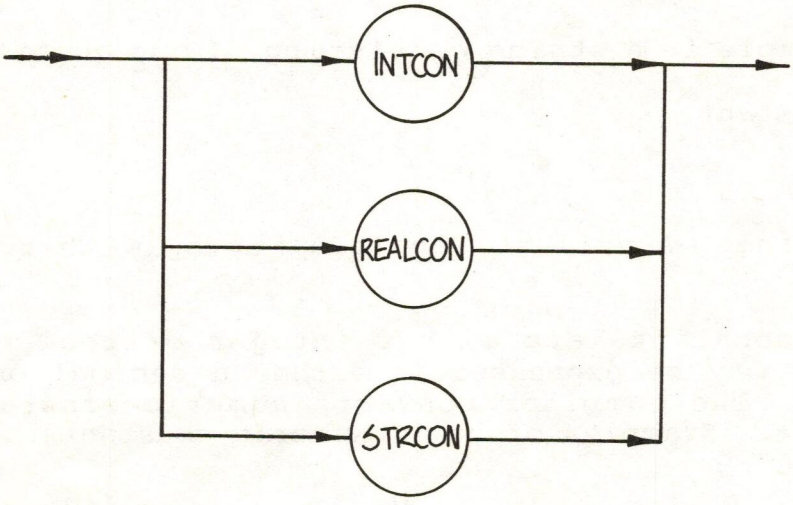
and

```
INVNO
```

are different identifiers.

1.1.3. Constants

A constant is a program element that does not vary during the execution of a program. Both string and numeric constants may be defined.



A string constant is a group or string of characters enclosed within quotation marks. The maximum number of characters allowed in a string constant is implementation dependent. See appendix E for current limits. In all cases at least 255 characters will be permitted.

Examples of valid string constants are:

"This is a valid string constant"

"ABC Development Company"

""

"PAYMENT DUE DATE: "

"..!#$%&'()=~^|\{[}]*:+;<..."

Two consecutive quotation marks within the string are treated as one quotation mark which is a part of the string. For example:

"He said ""The time has come"" before he left"

is the following string:

He said "The time has come" before he left

Other examples using imbedded quotation marks are:

""""

"this is a quotation mark """

The first example is a string consisting of one quotation mark.

The string constant:

""

is a null string. A null string is a string with a length of zero.

Numeric constants are either integer or real constants. Real constants may be expressed in either a decimal or floating point format. The compiler converts numeric constants to an internal format. Examples of valid numeric constants are:

1	Ø	32767
5478	12345	21
12.83	1267.	54.ØE Ø1
1.11E-21	Ø.Ø1E63	1.23E+61

A blank may appear following the E in a numeric constant. No other blanks may be used.

Integer constants are stored as two byte signed binary integers with a maximum magnitude of 32767. Real numbers are stored as eight byte binary coded decimal digits. The first byte is the sign and exponent; the remaining seven bytes represent the mantissa.

Numeric constants are always positive. If a sign is appended to a constant, the sign is treated as an unary arithmetic operator. See chapter five for a discussion of arithmetic operators.

The following numeric constants are invalid:

3.2E	missing exponent
1.23E+99	exponent out of range
12,734	commas are not permitted within constants
Ø.11.2	only one decimal point is permitted
12 .34	the blank is not permitted in the number

If a numeric constant does not contain a decimal point or an exponent the compiler will treat the constant as an integer unless the magnitude of the constant exceeds 32767, the maximum magnitude of CB8Ø integers. In this case the constant is treated as a real constant. In other words, 3ØØØØ is an integer but 3ØØØØØ is a real constant. 3ØØØØ.Ø is also a real constant.

Integer constants may also be expressed as hexadecimal or binary constants. A binary constant is a group of Ø's and 1's ending in the letter "B". Hexadecimal constants consist of a group of numeric characters and the letters "A" through "F". A hexadecimal constant ends with the letter "H".

In binary constants the letter B, and in hexadecimal constants the letters "A" through "F" and the letter "H", may be either lower-case or upper-case. The first character of a hexadecimal constant must be a digit.

The following list contains examples of valid binary and hexadecimal constants:

1100b	Ø1Ø1Ø1Ø1Ø1B	8ØØØh
7ABCH	7abch	1B
Ø7fffh	ØØØØH	ØFfFfH
Øh	111b	ØABCDH

Unlike decimal integer constants binary and hexadecimal constants will not be converted to real constants if their magnitude exceeds 32767. This allows bit patterns up to 16 bits long to be represented as constants. This means that while

65535

is treated as a real constant,

ØFFFFH

is a hexadecimal integer constant.

The following binary and hexadecimal constants are invalid:

fa3eh	does not start with a digit
7ABCD	missing H at end of constant
ØFFFFFH	exceeds the range of integers
Ø1Ø2Ø1b	binary contains digit other than Ø or 1
Ø 111 ØB	spaces not permitted in constants
1Ø11,1111	comma not permitted in constants

1.1.4. Remarks

Remarks are added to the source program to increase readability of the program. The compiler ignores remarks. If a remark is removed from a program there is no change in the code generated by the compiler. A remark starts with the reserved word REMARK or REM and is terminated with the physical end of the line unless it is continued to the next line with a backslash.

The following example shows valid remarks.

```
REM Any Characters

REMARK ACCOUNTS PAYABLE

REMARK \
       \ PAYROLL
       \ PROGRAMMED BY TIM SMITH
       \ LAST MODIFIED 28 JUNE 1981
         VERSION 1.Ø3
```

In the last example the backslashes have the effect of treating the five physical lines as one remark. Thus the backslash has specific meaning even as part of a remark. A remark may not contain a carriage return since the carriage return terminates the remark. The carriage return is not part of

the remark.

Remarks may appear anywhere in the source program with the following restrictions. A remark always terminates a statement. Statements are described in subsequent chapters. Remarks may not be imbedded in other program primitives. Finally remarks are not permitted as part of a DATA statement. DATA statements are explained in chapter 3.

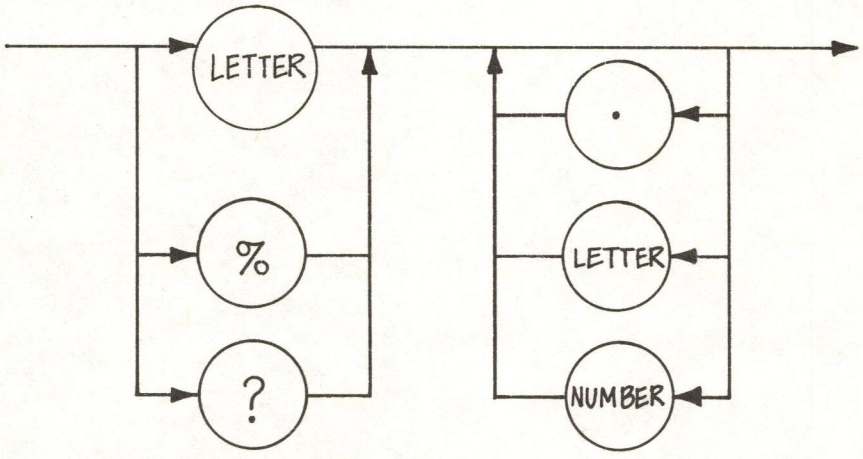
Any number of blank lines may be used within a program. A blank line is treated as a remark.

```
REM   THIS IS A REMARK CONTINUED \

      BUT THIS IS NOT PART OF THE REMARK
```

In the example above, the blank line becomes part of the remark but the third line is not a continuation of the remark.

1.2. Notation

The CB8Ø language manual uses syntax diagrams to show the syntax of each statement in the language. A syntax diagram shows the permissible constructs for each statement. For example the syntax diagram for an identifier is:



The rectangular box indicates a program element that is further defined by another syntax diagram. In this example a syntax diagram could be drawn to show that a letter is an A,B,C etc. The circle indicates a reserved symbol or token in the language. Arrows are used to represent the flow of control which indicates permissible alternative forms of the program element.

Program examples in this manual use upper case letter for both reserved words and identifiers. This is done for clarity. Any of the identifiers or reserved words could be written in lower case without altering the program.

The CB8Ø Language Manual is independent of the operating environment wherever possible. However, when file names must be shown, CP/M^{tm} file names are used. Digital Research's CP/M and its derivatives MP/M-8Ø^{tm} and CP/NET^{tm} are the standard operating systems for 8 bit microprocessors using CB8Ø. CP/M is a registered trademark,and MP/M-8Ø and CP/NET are trademarks of Digital Research, Pacific Grove, CA.

If CB8Ø is being used with an operating system other than CP/M the file specification may differ from those shown in this manual.

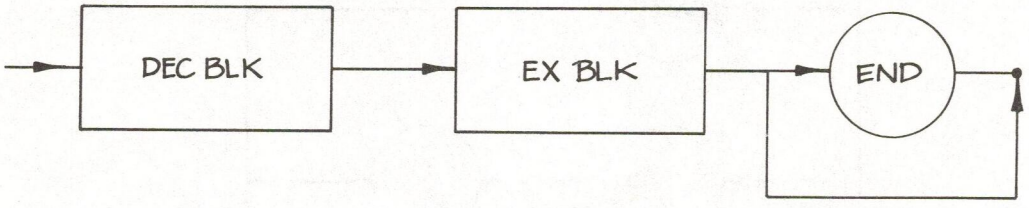
2. CB8Ø PROGRAM STRUCTURE

Chapter one defined program primitives from which all CB8Ø programs are built. This chapter describes the overall structure of CB8Ø programs. The structure is defined in terms of declaration and statement groups which will be explained in greater detail in subsequent chapters.

This chapter also describes compiler directives, which are used to provide information to the compiler.

2.1. CB8Ø Programs

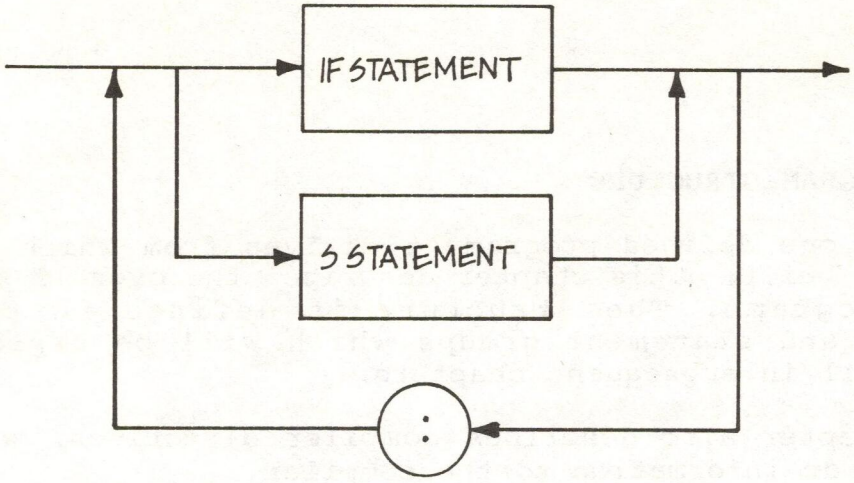
A CB8Ø program consists of a declaration group followed by a statement group. The program is terminated either by the reserved word END or when the end of the source program is reached. When an END is encountered in a source program, any text which follows the END is ignored.



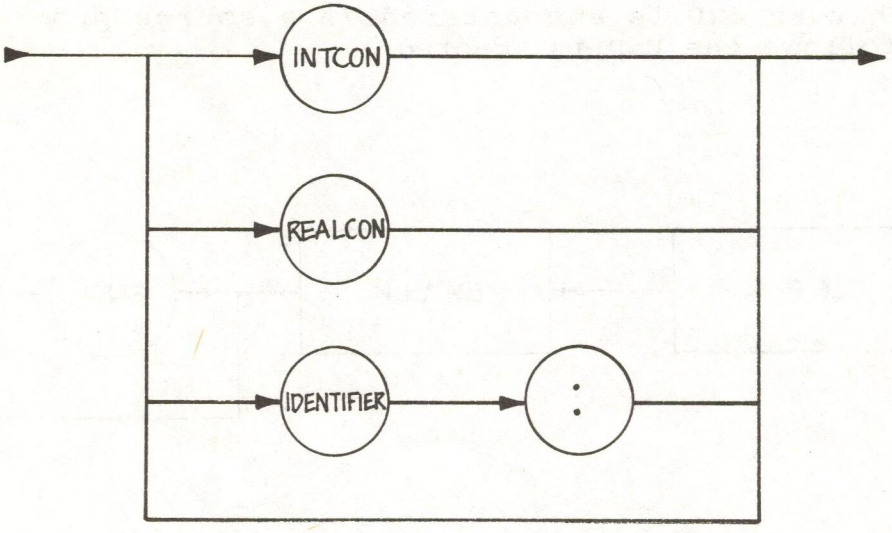
A declaration group consists of zero or more declaration statements. Declaration statements are explained in chapter 3.

A statement group consists of zero or more CB8Ø statements and multiple line functions. Multiple line functions are explained in chapter four. Chapters five through ten of this manual describe the CB8Ø statements.

A CB8Ø statement consists of an optional statement label, the statement proper and terminates with the end of a physical source line. With the exception of an assignment statement, all statements start with a reserved word.



A statement label may be an integer or real constant or an identifier with a colon appended to the end of the identifier.



When an identifier is used for a label it must not be used in another context within the program. This means it cannot be used as a variable or a function name. See chapter four for a discussion of local variables and labels within multiple line functions for an exception to this rule. A numeric constant used as a label may also be used as a constant within the program.

The following list contains valid statement labels:

1ØØ	23ØØ.ØØ	2222
GETRECORD:	PROCESS.COMMAND:	A:
2ØØEØ3	1ØØ.ØØ	Ø.ØØ1

The following statement labels are invalid:

1ØØH	hexadecimal constants are not permitted
XYZ	colon is missing
1#2	invalid constant
stop:	stop is a reserved word

When an alphanumeric label is referenced the colon is not part of the label. Chapter seven explains statements that reference labels.

When a numeric constant is used as a label, the characters making up the label determines the uniqueness of the label, not the value of the label itself. The labels 1ØØ.Ø and 1ØØ.ØØ are different labels even though they have the same numeric value.

Chapter one explained that the backslash character (\) is used by CB8Ø as a continuation character to allow statements to extend over many physical source statement lines. For example:

```
PRINT X, Y, Z
```

could be written as:

```
PRINT     \
      X, \
      Y, \
      Z
```

A continuation character causes all characters beginning with the continuation character and including the first end of line encountered to be ignored.

```
PRINT \ ALL THIS IS IGNORED
      X
```

A continuation character may appear anywhere that a blank may be used to separate program primitives. Thus the continuation character may separate two primitives:

```
PRINT \
        X
```

A continuation chracter may not split a primitive. The following example is an invalid use of the continuation character:

```
PRI\
NT X
```

A continuation character used within a string constant is treated as a character within the string. For example:

```
"AB\CD"
```

is a valid string constant with 5 characters.

Since all characters following the continuation character on the same physical line are ignored the space following a continuation character could be used to document a program.

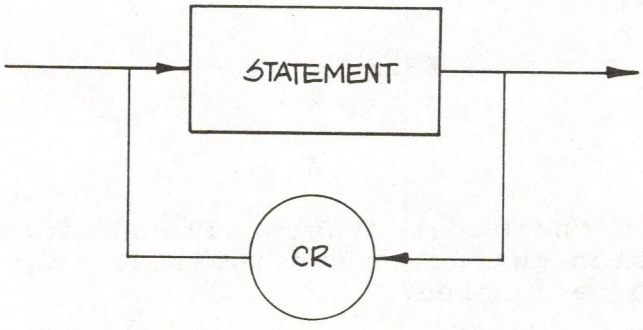
```
PRINT \   NOW PRINT THE TOTAL
    ACCOUNT.TOTAL
```

A remark terminates a statement. Thus the statement:

```
PRINT REMARK NOW PRINT THE TOTAL
    ACCOUNT.TOTAL
```

is not the same statement, and is in fact an incorrect CB8Ø statement.

At times it is necessary to associate a group of statements together. Normally this is used in conjunction with the IF statement described in chapter 6.



The special character colon (:) is used to indicate that two consecutive statements are part of a statement group. For instance:

```
PRINT X : PRINT Y
```

The colon must not be adjacent to an identifier to prevent confusion with a label.

All statements in a group must be part of one logical statement line. This means that if the statement group is spread over multiple source lines the continuation character must be used. For instance:

```
PRINT X :\
PRINT Y
```

would associate both statements in the same group. But

```
PRINT X :
PRINT Y
```

would not. In this last example the first line is a group of two statements consisting of a print statement followed by a null statement. The second line is another print statement not part of the statement group in the first line.

A colon serves to allow multiple statements on one line. In conjunction with the continuation character the colon allows groups or blocks of statements to be continued over many physical source lines.

2.2. Compiler Directives

Compiler directives are reserved words which are used to provide information to the compiler. They are not translated into executable code. All compiler directives begin with a percent sign (%). For instance:

```
%LIST
```

There may not be any blanks between the percent sign and the remainder of the directive. The compiler directive may appear anywhere within a source line but no other statements may appear on that line. Only blanks or tab characters may preceed the directive.

Any characters on the same line with the directive and following it are ignored unless they are required by the directive. A compiler directive may not be continued to another line with a continuation character, and it may not have a label.

2.2.1. Listing Control Directives

There are four compiler directives that affect the format of the listing product by CB8Ø. They are the %LIST, %NOLIST, %EJECT, and the %PAGE directives. Compiler toggles, explained in chapter twelve, also affect listings.

The %NOLIST directive stops listing the source file and interlisting code if that option is selected. The %LIST resumes listing the source file.

```
%LIST

%NOLIST
```

The %EJECT directive continues the listing on the top of the next page. The %EJECT directive is only in effect if the listing is being directed to the printer.

```
%EJECT
```

The %EJECT directive is also ignored if %NOLIST is in effect.

The %PAGE directive sets the page length of a listing directed to the printer. The desired length must be an integer constant following the %PAGE. For example:

```
%PAGE 4Ø
```

will set the page length to 4Ø lines.

2.2.2. %INCLUDE Directive

The %INCLUDE directive allows source code contained in an external file to be incorporated into the source program during compilation.

The included text is incorporated into the source directly after the %INCLUDE directive. The first character of the included text is treated as the next character in the source program. The physical line containing the %INCLUDE directive is not a part of the statement being compiled.

```
%INCLUDE CONDEF
```

The directive above will include source statements from the file CONDEF.BAS. The type extension of the file name defaults to "BAS". However, the programmer may specify any extension. For example:

```
%INCLUDE CONDEF.INC
```

The directive above will include the file CONDEF.INC. It is possible to specify that an include file be read from a drive other than the logical drive containing the source file. One method is to directly specify the drive.

```
%INCLUDE D:CONDEF.INC
```

Another method, which reads include files from a drive other than the one containing the source program, uses a compiler toggle. Compiler toggles are explained in section 12.2 of chapter 12.

Include files may be nested. The maximum depth of such nesting is implementation dependent. See appendix E for the current limitations. Programmers may assume that the maximum allowable depth will always be at least four, however some operating environments limit the number of files that may be open at one time.

A %INCLUDE directive may "split" a statement.

```
PRINT \
    %INCLUDE RECDEF.INC
```

If file RECDEF.INC contains the following source line:

```
NAME$
```

the %INCLUDE would have the effect of forming a source program with the following statement:

```
PRINT \
    NAME$
```

3. DATA TYPES AND DECLARATIONS

CB8Ø provides a variety of data types to support the requirements of programmers implementing commercial applications. A specific data item is either a constant or a variable. A constant is a data item that does not change value during execution of a program while a variable may assume different values during program execution.

There are three kinds of CB8Ø data: numeric, string, and label. The properties of these data items will be explained in the following sections.

3.1. Numeric Data

Numeric data falls into two classes: integer and real. Numeric data is used to represent arithmetic and logical quantities. Integer quantities are represented as two's complement binary numbers. Each integer requires two bytes for storage. If an integer is assigned a value outside the defined range of 15 binary digits (-32768 to 32767) the results will be undefined.

Integer data is processed more efficiently than real data because the hardware is designed to process integers directly. Integers should be used whenever possible to decrease execution time and to reduce the amount of memory used.

Real numeric data is stored as packed decimal digits in an eight byte floating point format. The first byte holds both the exponent and the sign of the number. The first bit is the sign of the number. The remaining 7 bits are the exponent.

The mantissa is seven bytes long and contains 14 digits. Values are always stored in a normalized format as 4 bit decimal digits. There are two digits stored in each byte of the mantissa.

The dynamic range of real numbers is 1.ØE-64 to 9.99999999999999E+62. Both the accuracy and dynamic range of CB8Ø numbers are significantly greater than that found in most binary implementations of real numbers.

The internal representation used by CB8Ø for some real numbers is shown below:

NUMBER	EXPONENT	MANTISSA
1.Ø	41H	ØØH ØØH ØØH ØØH ØØH ØØH 1ØH
-1.Ø	C1H	ØØH ØØH ØØH ØØH ØØH ØØH 1ØH
Ø.Ø123	3FH	ØØH ØØH ØØH ØØH ØØH 23H 1ØH
Ø.Ø	ØØH	(not significant if exponent byte Ø)
largest number	7FH	99H 99H 99H 99H 99H 99H 99H
smallest nonzero number	Ø1H	ØØH ØØH ØØH ØØH ØØH ØØH 1ØH

3.2. String Data

String data consists of variable length strings of characters. A string may have a maximum length of 32767 bytes. Space for strings is always allocated dynamically and released when the string is no longer required.

The first two bytes of a string represent the length of the string. The first byte of the length is the high order byte and the second byte is the low order byte. This is contrary to the normal storage of sixteen bit quantities in 8Ø8Ø microprocessors. The string "SAMPLE" is stored internally as:

LENGTH	BODY OF STRING
ØØH Ø6H	53H 41H 4DH 5ØH 4CH 45H

The left most bit (bit 7) of the first byte of the length is used by the system and must be ignored when accessing the string length. Thus the string length is actually the low-order 15 bits of the first two bytes of a string.

3.3. Label Data

Labels are always constants. They are used to reference statements and functions. Statement labels are explained in chapter 2; functions are explained in chapter 4.

Labels within the main executable block of a program must be unique. A label within a function (chapter 4) may be the same as a label in another function or the main executable block. Examples of labels are shown in chapter two.

3.4. Data Structures

CB8Ø supports two data structures. The first is simple variables. These are single values associated with a variable name. Simple variables may be of type integer, real, or string. For example the following identifiers represent simple variables:

```
AMOUNT          PAYMENT.DUE.DATE$       FIRST.FLAG%

INDEX%          ANGLE                   I
```

Integers are stored in two bytes of memory; real variables require eight bytes of storage. Strings are assigned two bytes of permanent storage which is used to store the address of the dynamically allocated string.

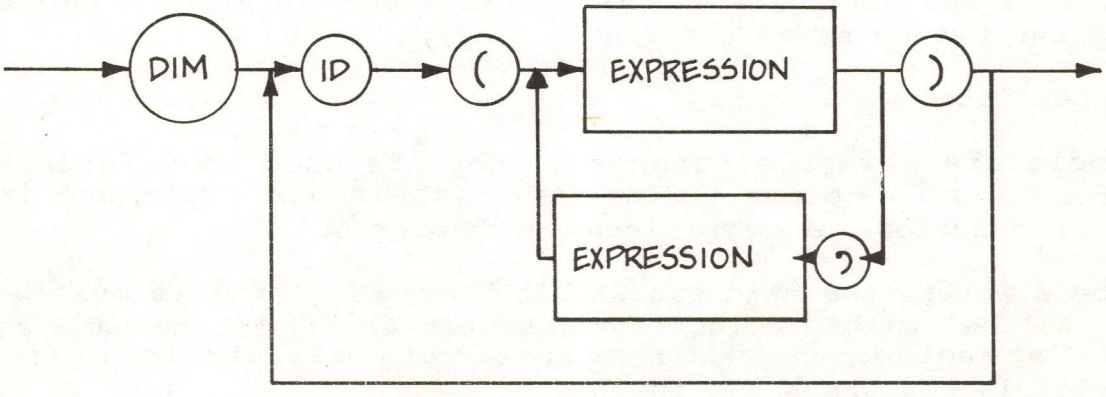
The other data structure provided by CB8Ø is arrays. An array associates a group of simple variables to one variable name. A particular element is identified by providing subscripts to select one variable in the array.

```
MATRIX(2,3)
```

MATRIX is the array name. The values in parenthesis are subscripts selecting a specific element of MATRIX. Since there are two subscripts, MATRIX is a two dimensional array.

Arrays may have any number of dimensions and the value of a dimension may be expressions determined during execution of the program. A particular implementation of CB8Ø may limit the number of dimensions allowed in an array. Refer to appendix E for current limitations.

The DIM statement causes space for an array to be dynamically allocated.



The expressions specify the upper bound for each subscript. Expressions are defined in chapter five. The lower bound is always zero. For example:

```
DIM X(25)
```

allocates an array with 26 elements, X(Ø), X(1), through X(25). The statement:

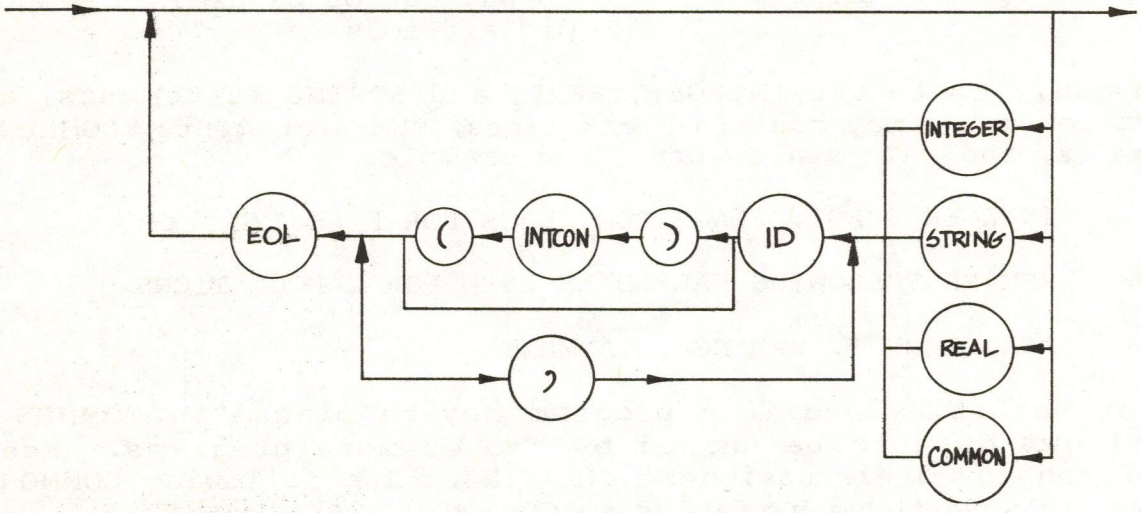
```
DIM ACCOUNTS(I,J)
```

creates space for (I+1) * (J + 1) elements.

The actual method of allocation is undefined in CB8Ø. CB8Ø does not define the order in which elements are stored in memory for a specific array. The method of allocation may vary from implementation to implementation. This approach is taken to allow allocation methods which allow efficient access to array elements on machines without hardware multiply.

3.5. Declarations

Declarations allow the programmer to specify that a specific variable or function name represents an integer, real, or string data type. Declarations are also used to place a variable in COMMON.



The following statements are valid declarations:

```
INTEGER I,J,LOOP.COUNT

REAL A, AMOUNT.DUE, C

STRING NAME,PART.DISCRIP
```

The statements above specify that identifiers I, J, and LOOP.COUNT represent integer data items and identifiers A,

AMOUNT.DUE, and C represent real data items. NAME and PART.DISCRIP are strings. If the identifier represents an array the number of subscripts is placed in parenthesis following the identifier name.

```
INTEGER MAX(2), Y(1)
```

The statement above declares MAX to be a two dimensioned integer array while Y has one dimension. This declaration does not result in allocation of space for the array. A DIM statement must still be executed prior to referencing any elements in the array.

Any statement in a declaration block may have a label. The label is ignored except that it is assigned the address of the first executable statement in the statement group that follows.

The following declarations are invalid:

`INTEGER I,J K`	missing comma
`REAL X(15,4Ø)`	arrays have number of dimensions in parenthesis
`STRING POS`	POS is a reserved word
`REAL X : INTEGER I`	colon cannot be used to group declarations

In addition to the INTEGER, REAL, and STRING statements, a declaration group may contain blank lines, REM statements, COMMON statements, and DATA statements. For example:

```
      INTEGER FLAG1, FLAG2 REM FLAGS FOR FILE I/O
1ØØ   REMARK FOLLOWING VARIABLES USED FOR CALCULATIONS
      REAL AMOUNT, BALANCE, PAYMENT
```

Any variable used in a program may be placed in COMMON. This allows data to be shared by two or more programs. See chapter ten for a discussion of CHAINING. The following COMMON statement places three variables in COMMON:

```
COMMON X, Y, Z
```

If the variable is subscripted then the number of subscripts is placed in parentheses following the variable name. For example:

```
COMMON A(2)
```

specifies that the variable A is a two dimensioned array.

The same variable may appear in a declaration statement and a COMMON statement. For example:

```
STRING X
COMMON X, Y
REAL   Y
```

There may be any number of COMMON statements in a declaration block. However if a declaration block is used in a multiple line function (chapter four) no COMMON statements may be included.

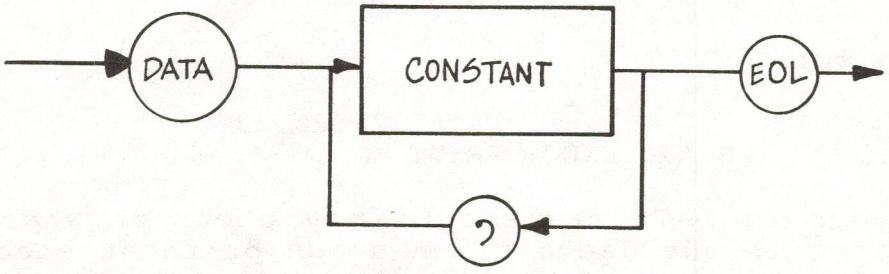
3.6. Default Declarations

CB8Ø provides default declarations for variables that do not appear in an INTEGER, REAL or STRING declaration statement. Variable names that end with a percent sign (%) default to integer variables, while variables ending in a dollar sign ($) default to string variables. Other variables default to real variables.

For example the variable X would be treated as a real variable while A$ is a string unless X and A$ appear in a declaration block INTEGER, REAL, or STRING statement.

3.7. DATA Statements

A DATA statement defines a list of constants which can be assigned to variables using a READ statement. READ statements are explained in chapter eight.



The following examples show valid DATA statements:

```
        DATA 1,2,3,4

100     DATA "APPLE", GRAPE, "ORANGE"

        DATA "$$$$$", "#####", """""""""""", \
             "!!!!!", "\\\\\"
```

The last example shows that a DATA statement may be continued to another line with the continuation character, but that backslashes may appear in string constants enclosed in quotation marks.

Strings do not have to be enclosed in quotation marks, but may be optionally delimited by commas. Whether or not a field is enclosed in quotation marks it must be terminated with a comma or a end of line character.

The following DATA statements are invalid:

```
DATA 12, ,13            missing field

DATA "ABC               missing quotation mark

DATA 1,2 REM VALUES     a REMARK not allowed here

DATA "AB"  "CD"         comma missing between strings
```

DATA statements may not appear in lines containing other statements. In other words a DATA statement may not be part of a statement group.

Labels are optional on DATA statements. Since a DATA statement is not executable but rather defines a list of constants that are available during execution, the label will actually address the first executable statement following the DATA statement. Thus the following example:

```
START.EXEC: DATA 1Ø,2Ø,3Ø
            PRINT X
```

is equivalent to:

```
            DATA 1Ø, 2Ø, 3Ø
START.EXEC: PRINT X
```

Any number of DATA statements may occur anywhere in a program, either in the declaration group or in an executable group. All DATA statements in a program, whether they occur as consecutive statements or not, are treated as one list of constants.

3.7.1. Identifier Usage

An identifier may not be used for two different elements even if the usage would not be ambiguous. An identifier used as a function name or as a label may not be used as a variable. In addition the same identifier may not be used as both a subscripted and non-subscripted variable.

The following example is invalid:

```
ACCOUNT: ACCOUNT = 3
```

The identifier ACCOUNT cannot be used as both a label and a simple variable.

The next example is also invalid:

```
X = X + X(3)
```

The identifier X cannot be used as both a subscripted and simple variable name.

Chapter four discusses the scope of variable names. It is possible for the same identifier to have two different uses when the scope of the identifiers is different.

4. USER DEFINED FUNCTIONS

A function allows the same group of statements to be executed from various points in a program. CB8Ø provides two types of functions: user defined functions described in this chapter, and predefined functions described in chapter six.

Functions may be included in the program that references it or they may be in separate modules. If the functions are in separate modules, each module is compiled and the modules are linked together.

4.1. Introduction to Functions

Functions perform operations which have limited and controlled interaction with the remainder of the program. CB8Ø supports two types of functions, single line and multiple line functions.

Both types of functions may have zero or more formal parameters. The definition of a function contains a list of the formal parameters that are assigned a value when the function is accessed. An actual parameter is an expression which is passed to the function when the function is referenced, and substitutes for a formal parameter.

When a function is accessed the number of formal and actual parameters must agree. In addition if the formal parameter is a string, then the actual parameter must evaluate to a string expression; if the formal parameter is numeric, the actual parameter must be numeric. An integer expression may be passed to a real formal parameter and an integer formal parameter may accept a real actual parameter. The appropriate conversion will occur.

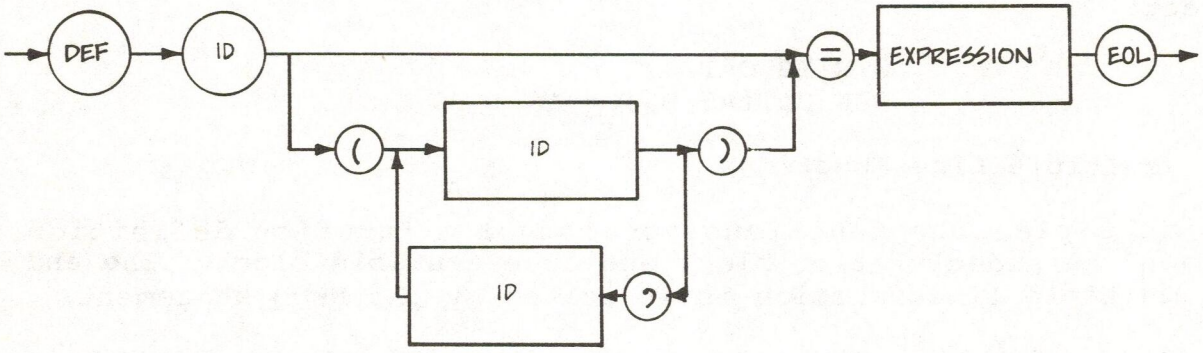
The maximum number of parameters allowed in a function may be limited by the implementation. See appendix E.

All parameters in CB8Ø are passed by value. This means that the actual parameter is evaluated before the function is executed. The value of the actual parameter is then passed to the function and becomes the initial value for the corresponding formal parameter. This method of passing parameters assures that changing a value of a formal parameter does not change the value of a variable outside the function.

Both single line and multiple line functions can be used as elements in an expression; a multiple line function can also be invoked through a CALL statement. CALL statements are explained in chapter seven.

4.2. Single Line Functions

Single line functions evaluate an expression and return the value of the expression. A single line function is similar to Fortran's statement function.



The ID is the function name. The expression may be any valid expression. Chapter five explains expressions. If the expression is of type string, the function name must be of type string.

The single line function is accessed by using its name in an expression. The following function calculates the average of two integers:

```
DEF AVERAGE%(A%,B%) = (A% + B%)/2
```

A% and B% are formal parameters. When the function is referenced actual parameters are substituted for the formal parameters and then the expression is evaluated.

The following statement uses the single line function AVERAGE% to determine the average of two expressions.

```
PRINT AVERAGE%(TEST.1% + 2,TEST.2%)
```

TEST.1% + 2 and TEST.2% are the actual parameters substituted for A% and B%.

The identifier used as a function name defines the type of value returned. The function AVERAGE% defined above returns an integer.

```
DEF CONVERT(A%) = A%
```

This function returns a real value since CONVERT is a real identifier.

```
DEF CAT$(A$,B$) = A$ + B$
```

The function CAT$ returns a string.

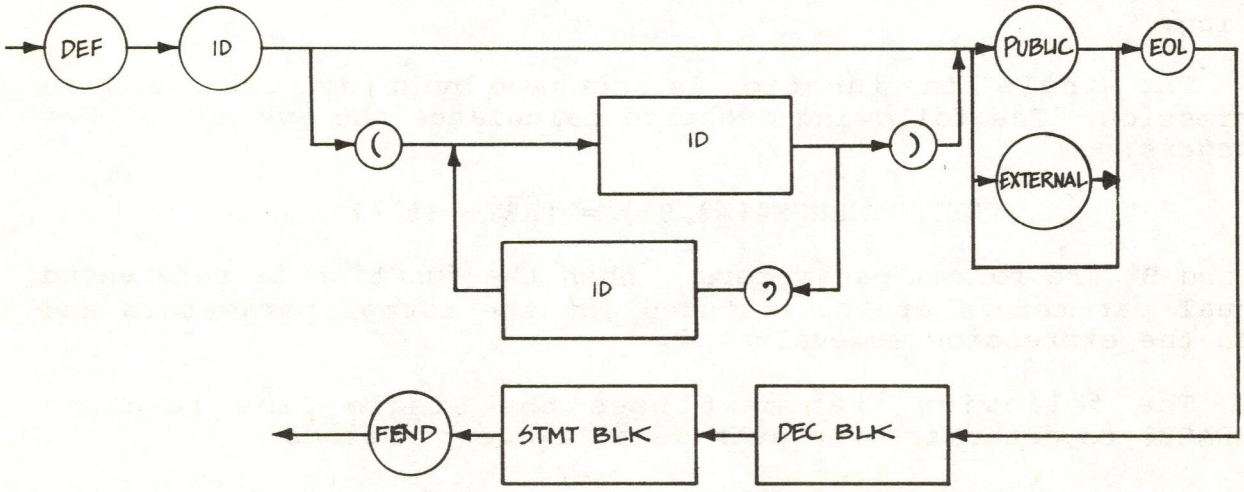
The names of single line functions may not appear in any declaration. For example the following statements are not correct:

```
STRING CAT
DEF CAT(A$,B$) = A$ + B$
```

4.3. Multiple Line Functions

Multiple line functions consist of a function definition followed by a declaration block and an executable block. The end of a multiple line function is indicated by the FEND statement.

Multiple line functions are equivalent to Fortran's subroutines and functions, or PL/I's procedures.



EXTERNAL and PUBLIC functions are explained in chapter eleven. They permit linkage with separately compiled modules.

```
DEF FN.NAME(F,M,L)
     STRING F,M,L,FN.NAME

     FN.NAME = F + " " + M + " " + L
FEND
DEF MEAN(X,Y)

     MEAN = (X + Y)/2.Ø
FEND
```

The declaration group may not contain a COMMON statement. Array variables may be declared but each execution of the DIM statement will result in a new array being dimensioned. Array names may not be passed as parameters; individual array elements may be used as actual parameters.

The executable block can contain any CB8Ø executable statements. However, a multiple line function may not contain another multiple line or a single line function. In other words function definitions may not be nested. In addition recursive references are not supported.

```
DEF LOOP(MAX)
     INTEGER MAX

     MORE:
          IF A < MAX THEN \
               A = A + 1 :\
               GOTO MORE
FEND

MORE:
     CALL LOOP
     GOTO MORE
```

This example shows a function with a local label MORE called by a program with a statement group using the same label MORE. The two labels are different; no confusion results from their use.

Multiple line functions are invoked either with a CALL statement explained in chapter seven, or by using the function as an element in an expression explained in chapter four. If the function is used as part of an expression, the function returns a value. The type of the value returned is the same as the type of the function name.

```
DEF A%
     .....
FEND

PRINT A%
```

The function A% returns an integer value. The value returned is the last value assigned to the function name prior to returning from the function. A function returns when the reserved word FEND is reached or when a RETURN statement is executed. RETURN statements are explained in chapter seven.

```
DEF GREATER(A,B)
     STRING GREATER, A, B

     IF A > B THEN \
          GREATER = A \
     ELSE \
          GREATER = B
     RETURN
FEND
```

The function GREATER will return a string which is equal to the greater of the two parameters. The function GREATER could also be called, with no value being returned. But in this example it would be of little practical value!

```
CALL GREATER
```

A RETURN statement in a function will result in a return from the most recently executed GOSUB or function reference. See chapter seven for a discussion of the GOSUB and RETURN statements.

4.4. Scope of Variables

All formal parameters and any variables declared in the declaration block are local to the function. In addition labels defined within a multiple line function are local to that function. This means that they are unknown or undefined outside the function.

```
INTEGER A,B,C,D
DEF TESTIT(A,B)
     INTEGER TESTIT,C

     C = A + B
     D = A / B
FEND
```

In the program above, the function TESTIT has 3 local variables. They are the formal parameters A and B, and the locally defined variable C. Note that the function name TESTIT is also declared as an integer within the function. The variables A, B, and C defined before the function are different variables from the three local variables A, B, C.

In the example above the variable D is not local to the function TESTIT. However, the value of D is accessed and in fact changed by TESTIT. A multiple line function may access and alter any variable that is available to the main program. That is, a variable which is not defined in a different multiple line function.

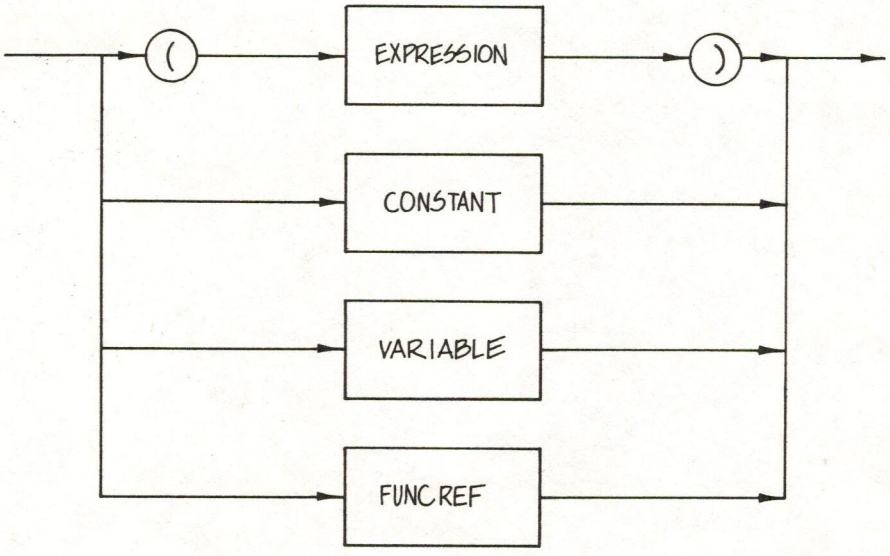
Changing D in the function TESTIT is a side effect of the function. These side effects can often cause unexpected results.

5. EXPRESSIONS AND ASSIGNMENTS

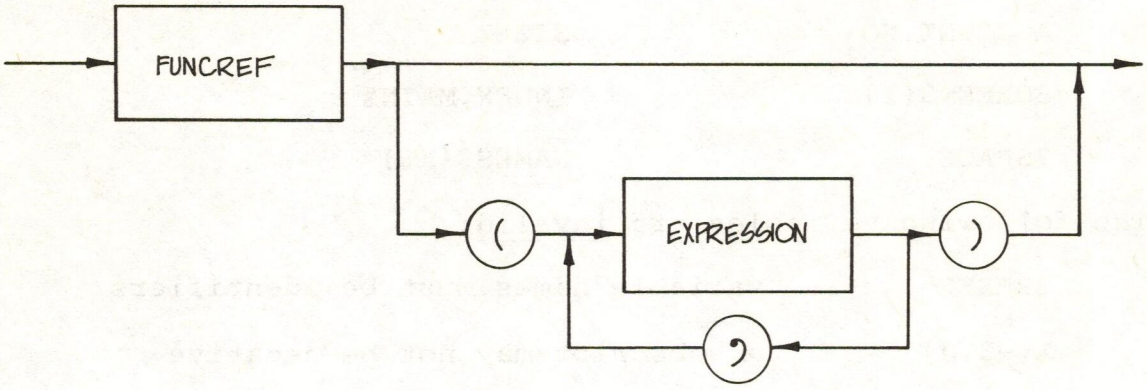
An expression is a combination of operands and operators that evaluate to a single value. Operands are variables, constants, or function references. The value of an expression may be saved by assigning it to a variable.

5.1. Operands

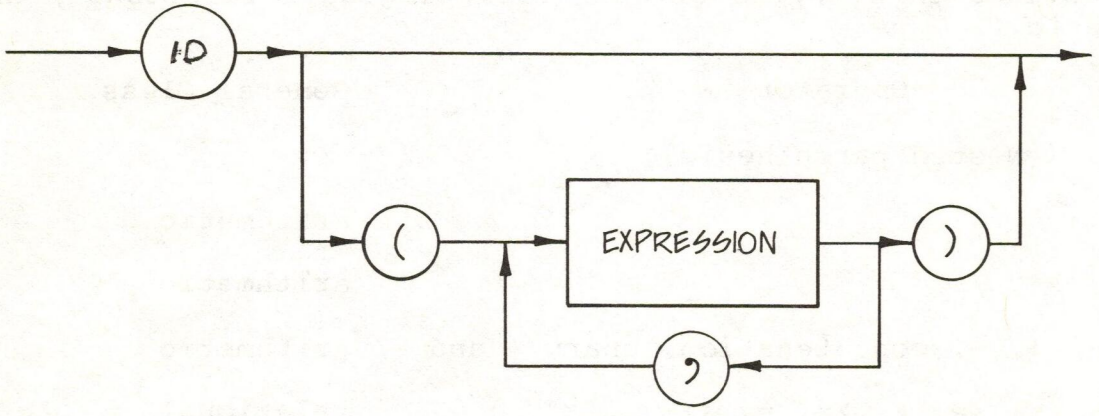
An operand is a variable, constant, function reference or an expression enclosed in parentheses.



Constants were discussed in chapter one. There are two types of functions: user defined functions and predefined functions. Accessing user defined functions was discussed in chapter four; predefined functions will be explained in chapter six.



This section will discuss accessing variables. A variable is a quantity that may change during program execution. Variables are assigned values by assignment statements, explained in this chapter, or by READ and INPUT statements, explained in chapter eight.



Variables may be simple variables or subscripted variables. A subscripted variable selects a specific element in an array. Before an array element may be accessed, space must be allocated for the array using a DIM statement.

The value of a variable is the last value assigned to that variable. If no value has been assigned to a variable, the value is undefined. Some implementations may assign initial values to variables but this is not required. Refer to appendix E.

The following list shows valid variables:

X	MAT(I,3)
ACCOUNT.NO	SIZE%
SCREEN$(I)	INDEX.MAIN%
?SPACE	NAMES$(K%)

The following variables are invalid:

3RRRRR	variable names must be identifiers
X(-3,J)	a subscript may not be negative
FINISH:	this is a label
STOP	a reserved word may not be a variable

5.2. Operators

Operators perform prefix and infix operations on operands. CB8Ø provides three types of operators: logical, relational, and arithmetic.

Operator	General Class
(Nested parenthesis)	
^	arithmetic
*, /	arithmetic
+, -, concatenation, unary + and -	arithmetic
<, <=, >, >=, =, <>	relational
NOT	logical
AND	logical
OR, XOR	logical

The precedence of operators in CB8Ø is listed above. A higher precedence operator will be evaluated before a lower precedence operator. For example the expression:

X + Y * Z

will be evaluated by first multiplying Y by Z and then adding the result to X. This is because multiplication (*) has a higher precedence than addition (+).

```
X / Y * Z
```

In the expression above the division will be performed first because multiplication and division are of equal precedence. By using parenthesis the order of evaluation may be altered.

If the type of two operands differ, conversion to a common type is required. The following table lists the rules for converting operands.

	REAL	INTEGER	STRING
REAL	N/C	REAL	ERROR
INTEGER	REAL	N/C	ERROR
STRING	ERROR	ERROR	N/C

N/C indicates that no conversion is required; ERROR indicates that operands of those types cannot be used together.

Concatenation (+) combines or adds together two strings. It is the only arithmetic operator that may be used with strings.

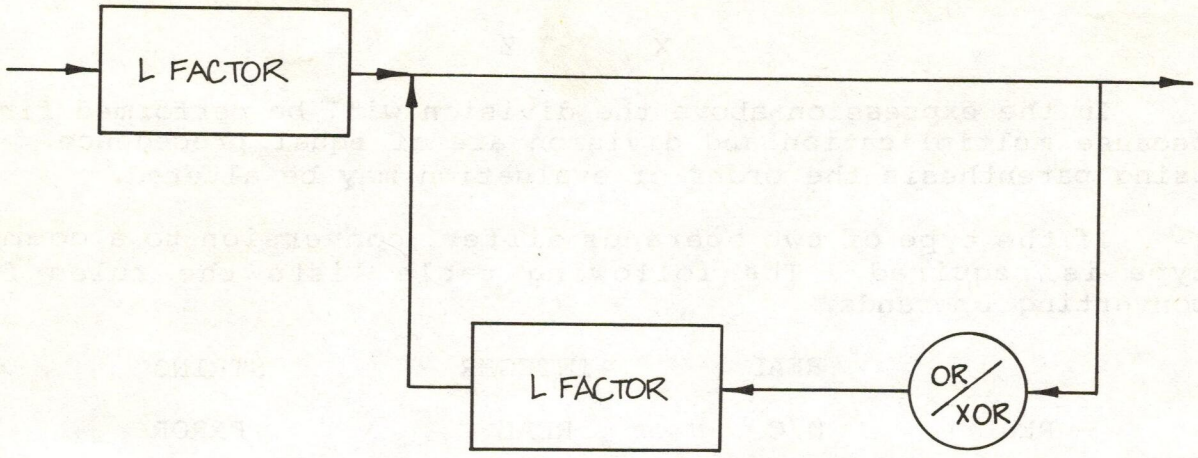
5.2.1. Logical Operators

CB8Ø provides logical operators AND, OR, XOR, and NOT. NOT is a prefix operator, the others are infix operators. All logical operators require numeric operands. If the type of an operand is real it is converted to an integer. All logical operators treat an operand as a 16 bit binary quantity.

The logical OR and XOR operators require two operands and perform a bitwise OR or XOR operation on the operands. The tables below define the OR and XOR operators:

OR	Ø	1
Ø	Ø	1
1	1	1

XOR	Ø	1
Ø	Ø	1
1	1	Ø



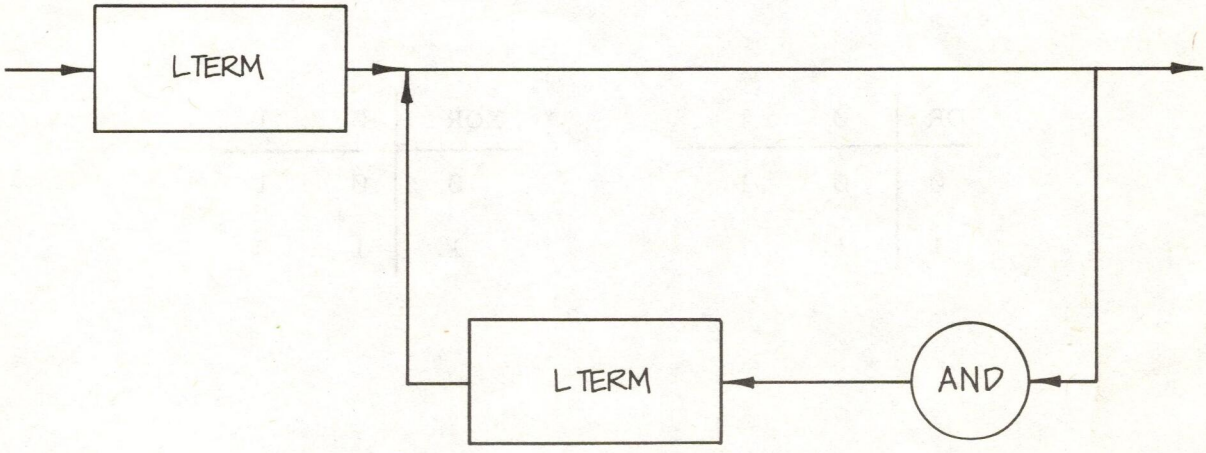
The OR operator is often used to "turn on" or set bits in an integer variable. For example:

```
FLAG% OR 7ØØH
```

will insure that bits 9 through 11 are a 1 (on). The notation used for defining bits is that the least significant bit is bit Ø and the most significant bit is bit 15.

The logical AND operator requires two operands and performs a bitwise AND operation on the operands. The table below defines the AND operator.

AND	Ø	1
Ø	Ø	Ø
1	Ø	1



The AND operator may be used to "turn off" bits in an integer. For example:

```
FLAG% AND 8ØFFH
```

will insure that bits 9 through 14 are Ø (off).

The logical NOT operator requires one operand. The NOT operator inverts each bit of the operand. This results in the 1's complement of the operand.

The syntax diagram for the NOT operator is shown as part of the syntax diagram for relational operators.

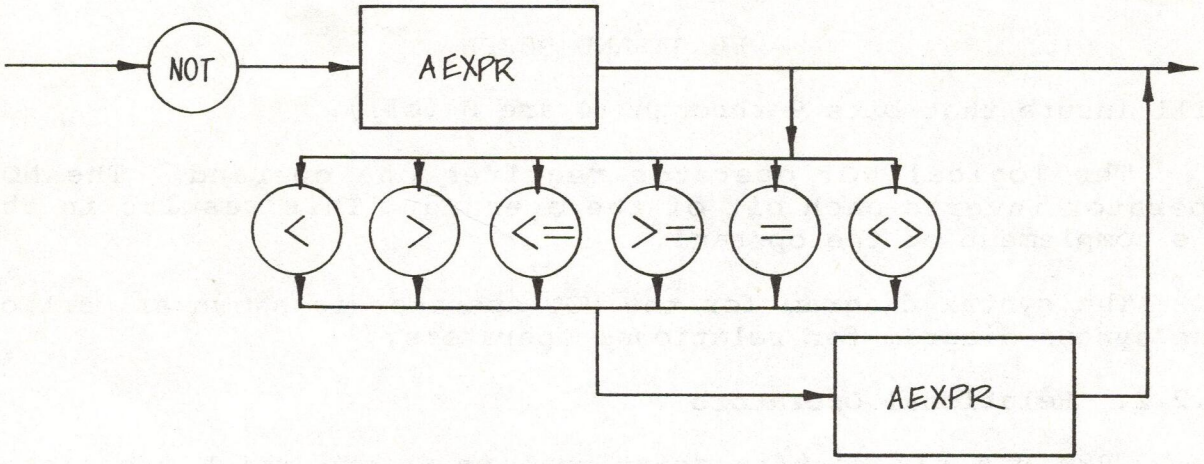
5.2.2. Relational Operators

CB8Ø has six postfix relational operators which are listed in the table below. Relational operators compare two operands producing an integer result. If the relationship is true the result is a negative one (all 1 bits) otherwise it is a zero.

OPERATOR	RELATION
<	LESS THAN
<=	LESS THAN OR EQUAL
>	GREATER THAN
>=	GREATER THAN OR EQUAL
=	EQUAL
<>	NOT EQUAL

The value resulting from a relational operator is either true, the relationship holds, or false, the relationship does not hold. True is a value of ØFFFFH, and false is zero. This means that not true is false!

Expressions containing relational operators are most frequently used with WHILE loops and IF statements. See chapter seven for a description of the IF and WHILE statements.



The operands must both be numeric or both be of type string. If one operand is real and the other is an integer, the integer is converted to a real value before performing the comparison.

```
A < B

ANSWER$ = "STOP"

(I% <= J%) OR (X > Y)

INDEX% <> ANGLE
```

The examples above show relational operators with real, string, and integer operands. In each case the result of the operation is an integer value.

The following expressions show invalid use of relational operators:

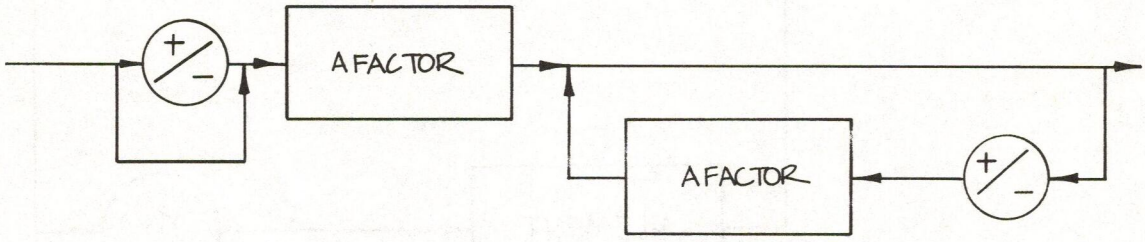
`A$ < B%`	cannot compare a string and an integer
`I% >< B%`	not a valid relational operator
`X NOT = Y`	invalid syntax (use <>)

5.2.3. Arithmetic Operators

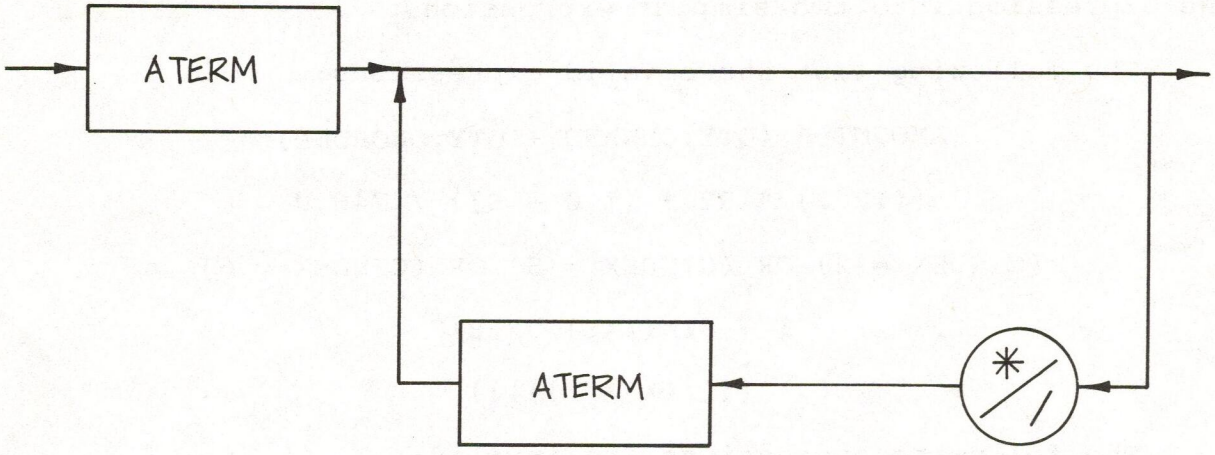
Five arithmetic operators are provided by CB8Ø: addition, subtraction, multiplication, division, and exponentiation. Addition and subtraction may be used as prefix or infix operators; the others may only be used as infix operators.

Addition and subtraction can be performed on both integer and real operands. If one operand is real and the other is an integer, the integer is converted to a real value. String

variables may be concatenated together using the infix operator for addition (+).

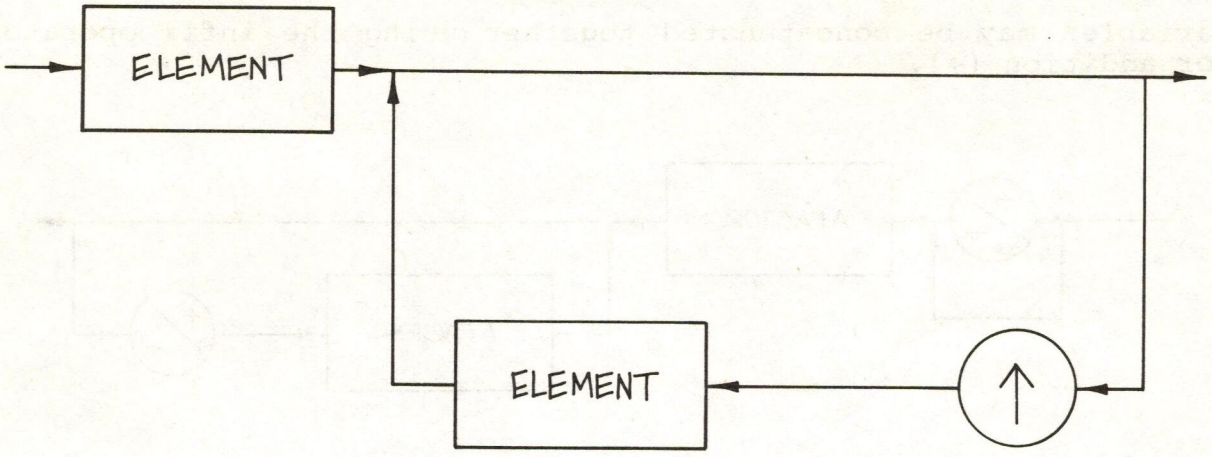


Multiplication and division can be performed on both integer and real operands. If one operand is real and the other an integer, the integer is converted to a real value.



The final arithmetic operator is exponentiation, which is also performed on both integer and real operands. The first operand is raised to the power represented by the second operand. If one operand is real and the other is an integer, the integer is converted to a real value.

A negative real value cannot be raised to a power. An execution error will occur if the operand on the left of the operator is negative.



Operators may be combined into complex expressions. For any implementation there will be a limit on the complexity of expressions. This should not affect most programs. If a compiler error occurs because an expression is too complex, break the expression into two simpler expressions.

The following list shows valid expressions:

```
AMOUNT * (QTY.ONHAND + QTY.ONORDER)

((I2^2) * R2 * (1.Ø - S)) / 746.Ø

(CINDEX = 2) OR (CINDEX = 5) OR (CINDEX = 6)

I + SIN(X<Y) OR B/C

(((((X + Y)))))
```

The following expressions are <u>invalid</u>:

X + A$	invalid operands (string and real)
I% - J% K%	operator missing between J% and K%
- A$	unary minus not allowed with string operand
(X * Y))	parentheses are not matched

It is possible for all arithmetic operators to overflow the maximum magnitude permitted for the type of operand involved. For example:

```
INTEGER X,Y,Z

X = 3ØØØØ
Y = 3ØØØØ
Z = X + Y
```

In the example above the additon would overflow the maximum magnitude of 32767 allowed for integer values. If the operands are integers, overflow is ignored. If the operands are real values, an execution error occurs when an overflow is detected.

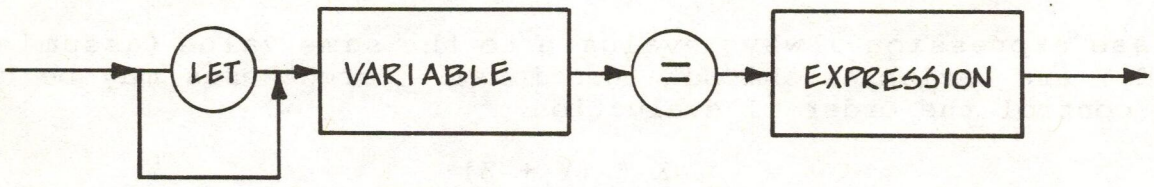
String overflow will also cause an execution error. For example, if two strings, each with a length of 2Ø,ØØØ characters, were concatenated together the new string would have to be 4Ø,ØØØ characters long. This is greater than the maximum string length and would result in an execution error.

Division of a real value by zero results in an execution error, but division of an integer by Ø produces an undefined result.

Overflow of integer calculations is not checked because of the substantial reduction in performance which would result on eight bit microprocessors if such checks were made. A particular implementation may check for these conditions.

5.2.4. Assignment Statements

The assignment statement sets a variable equal to the value of an expression.



The value of the expression is assigned to the variable at the left of the equal sign (=).

```
LET X = Y + X

LET A$(I,J) = B$ + C$
```

The reserved word LET is optional; normally it is not used.

```
X = A + 1.Ø
```

If the type of the variable on the left of the equal sign is a string, the expression on the right must evaluate to a string. When the variable is numeric, the expression must also be numeric. The expression will be converted to the type of the variable, either integer or real, if required.

```
A$ = B$ + C$(I%)

LET X = W * Y + 1.Ø

I% = X
```

The last expression will cause the variable X to be converted to an integer, and then assigned to the variable I%. If a real value is greater than the maximum magnitude of integers the result of the conversion is undefined.

The following assignment statements are invalid:

`A$ = X + 1`	numeric expressions cannot be assigned to a string variable
`X,Y = A + 1`	only one variable is allowed on the left of the equal sign

5.2.5. Evaluation of Expressions

Expressions are evaluated in a manner that ensures the hierarchy of operators is preserved and that normal algebraic properties (such as commutativity) are retained.

```
X + Y and Y + X
```

These expression always evaluate to the same value (assuming X and Y are variables and not functions). Parentheses may be used to control the order of evaluation.

```
X * (Y + Z)
```

The expression above performs the addition of Y and Z prior to multiplying by X. But the expression:

```
X * Y + Z
```

performs the multiplication first.

In order to provide the maximum opportunity for optimization, no other order of evaluation is implied. In particular, if operations are commutative, CB8Ø may use this property to rearrange the expression. This could result in two different implementations giving different values to the same expression. This would normally be caused by side effects resulting from the evaluation of functions.

```
DEF SAMPLE

        W = 2
        X = 4
FEND

W = 1
Y = A + X + W
Z = A + W + X
```

In this example, the side effect of setting W equal to 2 in the function X causes Y and Z to have different values.

5.3. Mixed Mode Expressions

Mixed mode expressions are expressions in which an infix operator has an integer and a real operand. In general mixed mode expressions generate more code and execute more slowly than expressions that do not use mixed mode.

```
A = X + Y%
```

The assignment above has a mixed mode expression. The operand X is real, and the operand Y% is an integer. The expression:

```
X = X + 2
```

is also mixed mode since the constant 2 is an integer constant. If the expression had been written as:

```
X = X + 2.Ø
```

it would not be mixed mode. These last two examples are an exception to the rule that mixed mode generates more code. In these examples the first expression generates less code than the second one since the real constant (2.Ø) takes eight bytes to store.

6. PREDEFINED FUNCTIONS

Predefined functions return a value to be used as an operand in an expression. The type of the actual parameters of a function must match with the usual convention that integer and real values may be used interchangeably.

In this chapter, when an X is used as a parameter it represents a real numeric expression. I% represents an integer expression, while a A$ is a string expression.

6.1. Numeric Functions

6.1.1. ABS

The ABS function returns the absolute value of the argument. The argument must be numeric and will be converted to a real value if it is an integer. ABS returns a real value.

ABS(X)

6.1.2. ATN

The ATN function returns the arc-tangent or inverse-tangent of the argument. The argument must be numeric and will be converted to a real value if it is an integer. ATN returns a real value.

The ATN function is calculated using Chebyshev polynomials for maximum accuracy.

The argument X is expressed in radians.

ATN(X)

6.1.3. COS

The COS function returns the cosine of the argument. The argument must be numeric and will be converted to a real value if it is an integer. The COS function returns a real value.

The COS function is calculated using Chebyshev polynomials for maximum accuracy. The argument X is expressed in radians.

COS(X)

6.1.4. EXP

The EXP function returns the irrational constant "e" raised to the power of the argument. The argument must be numeric and will be converted to a real value if it is an integer. EXP returns a real value.

The EXP function is calculated using Chebyshev polynomials for maximum accuracy.

```
EXP(X)
```

6.1.5. FLOAT

The FLOAT function returns a real value equivalent to the integer argument. The argument must be numeric and will be converted to an integer if it is a real value.

```
FLOAT(I%)
```

6.1.6. INT and INT%

The INT and INT% functions convert their arguments to whole numbers. The argument must be numeric and will be converted to a real value if it is an integer. Both functions truncate the argument to a whole number.

The INT function returns a real value, while the INT% function returns an integer value.

```
INT(X)

INT%(X)
```

6.1.7. LOG

The LOG function returns the natural or Naperian logarithm of the argument. The argument must be numeric and will be converted to a real value if it is an integer. LOG returns a real value.

The LOG function is calculated using Chebyshev polynomials for maximum accuracy.

```
LOG(X)
```

6.1.8. MOD

The MOD function returns the remainder after dividing the first parameter by the second parameter. Both arguments must be numeric and will be converted to an integer value if either is a real value. MOD returns an integer value.

```
MOD(I%,J%)
```

6.1.9. SGN

The SGN function returns an integer value that represents the algebraic sign of the argument. SGN returns a -1 if the argument is negative, Ø if it is Ø, and a positive 1 if the argument is positive.

The argument must be numeric and will be converted to a real value if it is an integer.

SGN(X)

6.1.1Ø. SIN

The SIN function returns the sine of the argument. The argument must be numeric and will be converted to a real value if it is an integer. SIN returns a real value.

The SIN function is calculated using Chebyshev polynomials for maximum accuracy.

The argument X is expressed in radians.

SIN(X)

6.1.11. SQR

The SQR function returns the square root of the argument. The argument must be a numeric value; it will be converted to a real value if it is an integer. If the argument is negative an execution error will occur. SQR returns a real value.

The SQR function is calculated using Newton's method.

SQR(X)

6.1.12. TAN

The TAN function returns the tangent of the argument. The argument must be numeric and will be converted to a real value if it is an integer. TAN returns a real value.

The TAN function is calculated using the following identity:

TAN(X) = SIN(X) / COS(X)

The argument X is expressed in radians.

6.2. String Functions

6.2.1. ASC

The ASC function returns the ASCII numeric value of the first character of the string argument. The value returned is an integer.

```
ASC(A$)
```

6.2.2. CHR$

The CHR$ function returns a one character string which is the ASCII character represented by the value of the argument modulo 256. The argument must be numeric; if it is a real value it is converted to an integer.

```
CHR$(I%)
```

6.2.3. LEFT$

The LEFT$ function returns a string which includes the left most characters of the first argument. The length of the string returned is the lesser of the length of the first argument and the value of the second argument.

```
LEFT$(A$,LEN%)
```

The second argument must be numeric; if it is a real value it is converted to an integer. If the second argument is zero, a null string is returned. If the second argument is negative, an execution error occurs.

```
LEFT$("ABC",2) returns "AB"
```

If the second argument is longer than the length of the first argument, the first argument is returned.

```
LEFT$("ABC",5) returns "ABC"
```

6.2.4. LEN

The LEN function returns the length of the string argument. If the argument is a null string, zero is returned.

```
LEN(A$)
```

6.2.5. MATCH

The MATCH function has three arguments. The first one is a pattern string, the second is the target string, and the final parameter is a numeric value.

The MATCH function returns the position of the first occurrence of the pattern string in the target string or zero if no match is found. Searching starts at the position in the target string determined by the third parameter.

If the third parameter is real it is converted to an integer. If the third parameter is zero or negative an execution error occurs.

A zero is returned if either the pattern string or the target string is a null string.

```
MATCH(PATTERN$,TARGET$,I%)
```

The MATCH function provides special pattern characters for matching different classes of characters. The following table provides a list of these characters.

PATTERN	CLASS OF CHARACTERS IT MATCHES
#	match any digit
!	match any lower-case or upper-case letter
?	match any character

For example:

```
MATCH("##","ABC1A123",1) returns a 6

MATCH("##","ABC1A123",7) returns a 7

MATCH("?!#","3 people are in A1",1) returns a 16
```

If a backslash (\) precedes a character in the pattern string and the next character is a #, !, or ? the special meaning defined in the table above is ignored. In other words the backslash is used as an escape to override the special pattern matching characters.

Thus

```
MATCH("ABC\#","12ABC#",1) returns a 3
```

but

```
MATCH("ABC#","12ABC#",1) returns a Ø
```

6.2.6. MID$

The MID$ function returns a string which is a segment of the first argument. The segment starts with the character position represented by the second argument. The third argument is the length of the segment.

MID$(A$,START%,LEN%)

The second and third arguments must be numeric; if they are real they will be converted to integers. If the third argument is zero, a null string is returned.

MID$("ABCD",2,2) returns "BC"

If the second argument is zero or negative, or if the third argument is negative an execution error occurs.

If the second argument is greater than the length of the first argument a null string is returned.

MID$("ABCD",5,3) returns a null string

6.2.7. RIGHT$

The RIGHT$ function returns a string which includes the right most characters of the first argument. The length of the string returned is the lesser of the length of the first argument and the value of the second argument.

RIGHT$(A$,LEN%)

The second argument must be numeric; if it is a real value it is converted to an integer. If the second argument is zero, a null string is returned. If the second argument is negative, an execution error occurs.

RIGHT$("ABC",2) returns "BC"

If the second argument is longer than the length of the first argument, the first argument is returned.

RIGHT$("ABC",5) returns "ABC"

6.2.8. STR$

The STR$ function converts the numeric argument to a string which is an ASCII representation of the number.

The argument must be numeric, if it is an integer it will be converted to a real value.

STR$(X)

The number is converted to a string in the same manner as unformatted output is printed to the console. See chapter eight. The only difference between the string returned by STR$ and the string printed to the console is that STR$ removes all blanks from the number.

6.2.9. UCASE$

The UCASE$ function returns a string in which the lower-case characters in the argument have been translated to upper-case. Other characters are not altered.

```
UCASE$(A$)
```

The argument remains unchanged unless it is set equal to UCASE$(A$).

```
A$ = UCASE$(A$)
```

alters the argument A$ but the assignment:

```
B$ = UCASE$(A$)
```

does not change A$.

6.2.1Ø. VAL

The VAL function converts the argument into a floating point number. Conversion is identical to that used to input characters from the console. See chapter eight.

If the argument is a null string, zero will be returned.

```
VAL(A$)
```

6.3. Miscellaneous Functions

6.3.1. COMMAND$

The COMMAND$ function returns a string equal to the command line that was used when the program was executed. The command line does not contain the name of the program executed and has had leading blanks removed. All lower-case letters are translated to upper-case.

```
COMMAND$
```

Some operating systems may require that COMMAND$ be implemented differently.

6.3.2. ERR

The ERR function returns a string equal to the last execution error that occurred. The string returned will be two characters in length. Appendix D lists the possible execution error codes.

The ERR function is intended to be used in conjunction with the ON ERROR statement explained in chapter seven.

```
ERR
```

If no error has occurred, the ERR function returns a null string.

6.3.3. ERRL

The ERRL function returns the line number of the last physical source line executed. The value returned by ERRL is an integer.

The source program must be compiled with the N toggle otherwise a zero is always returned.

```
ERRL
```

6.3.4. FRE

The FRE function returns the total amount of unallocated or free dynamic memory space. FRE returns a binary value. This means that when the value FRE returns is negative it represents a large positive number!

```
FRE
```

When using the value returned by FRE, care must be taken to insure that "negative" values are interpreted correctly. In general, if FRE returns a value which is negative, there is ample space remaining in dynamic memory space. The following statement may be used to determine that dynamic memory is at a low level.

```
IF (FRE > Ø) and (FRE < MIN.MEMORY%) THEN \
        CALL LOW.MEMORY.WARNING
```

This also applies to the MFRE function described below.

6.3.5. MFRE

The MFRE function returns the largest contiguous area of dynamic memory that is available. MFRE returns an integer value.

```
MFRE
```

MFRE will always return a value less than or equal to FRE.

6.3.6. SADD

The SADD function returns an integer value which is the address of the string argument. The address returned is a 16 bit quantity ranging from Ø to 65535. A zero value means that the argument was a null string. A null string may also have a zero length.

```
SADD(A$)
```

The SADD function will not accept an expression as an argument.

6.3.7. VARPTR

The VARPTR function returns an integer value which is the permanent storage space assigned to the argument. The argument may be an integer, real, or string variable.

```
VARPTR(<VARIABLE>)
```

The VARPTR function will accept the following arguments:

Name of a simple variable	VARPTR(X)
Name of a subscripted variable	DIM A$(1Ø) VARPTR(A$)
Element of an array	VARPTR(I%(2))

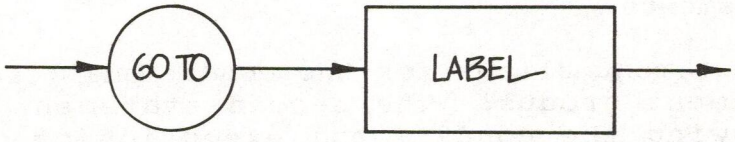
The argument may be an integer, real, or string variable. VARPTR will not accept an expression as an argument.

7. FLOW OF CONTROL STATEMENTS

Normally program statements are executed in the order in which they occur in the program. This chapter describes statements which allow the execution sequence to be altered.

7.1. GOTO Statements

The GOTO statement causes execution to be transferred to a statement label specified by the GOTO statement.



The label referenced must be defined within the program but need not be defined before it is used in the GOTO statement.

If the GOTO statement is part of the executable group of a multiple line function then the referenced label must be contained within that function. Likewise a GOTO statement outside a function cannot refer to a label within the body of the function. In other words a label within a function is local to that function. Its existence is unknown outside the function.

As explained in chapter two, if an alphanumeric label is referenced, the colon is left off.

```
X: GOTO X
```

If the label referenced in a GOTO statement is not part of an executable statement, the next executable statement after the label is executed. In the example below the REM statement is not an executable statement. Thus executing the GOTO 1ØØ would result in the PRINT statement being the next statement executed.

```
100  REM THIS IS NOT EXECUTED
     PRINT X
     GOTO 100
```

The following examples show valid GOTO statements:

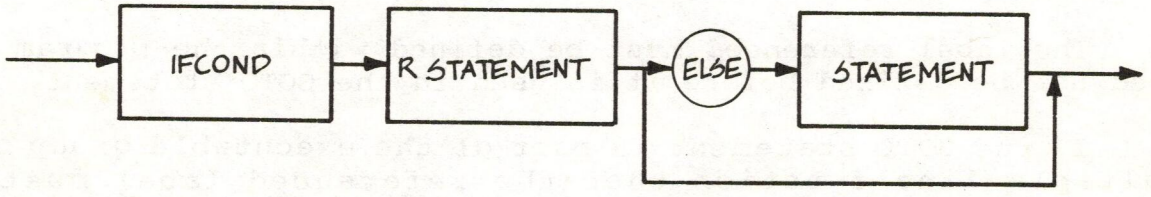
```
GOTO 100

GOTO START.OVER

GOTO 100E-01
```

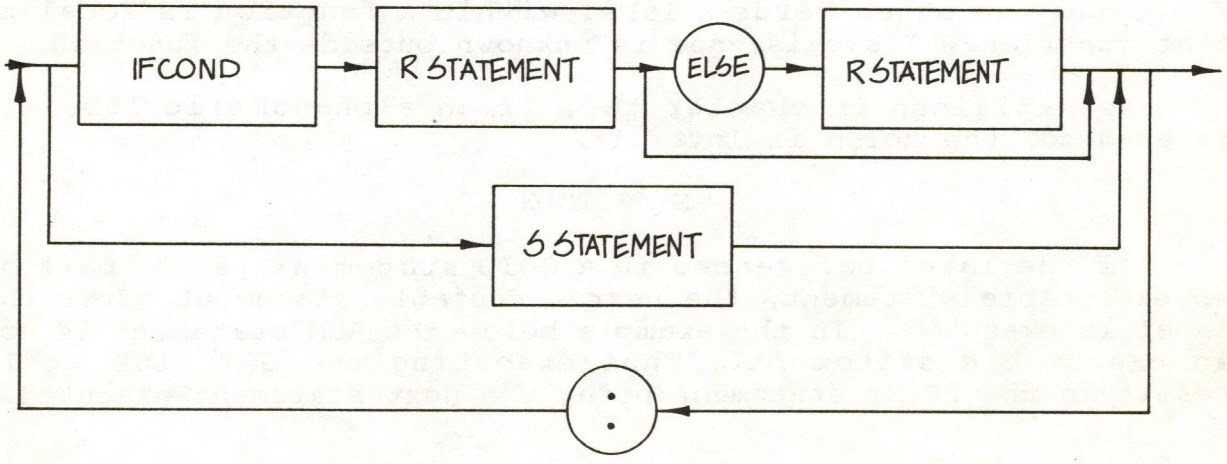
The following GOTO statements are invalid:

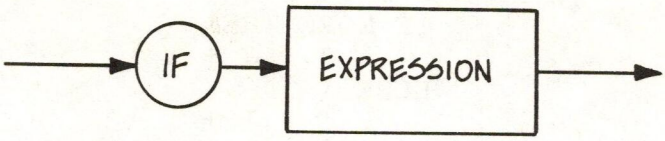
GOTO BEGIN:	colon is not part of the label referenced
GOTO ØFFFFH	hexadecimal constants cannot be labels
GOTO STOP	a reserved word cannot be a label

7.2. IF Statements

An IF statement allows for the conditional execution of one of two statement groups. The second statement group may be omitted allowing the conditional execution of one statement group.







The expression following the reserved word IF must be a numeric expression. If the value of the expression is zero (Ø) the expression is considered false; any other value is true. In other words the expression is a "logical expression", having either a true or false value.

When the logical expression is true the first statement group is executed. For example:

```
A = 2
B = 3
IF A < B THEN \
     PRINT "FIRST GROUP EXECUTED" \
ELSE \
     PRINT "SECOND GROUP EXECUTED"
```

In this example "FIRST GROUP EXECUTED" would be printed since the value of A is less than the value of B. If the expression had been false, "SECOND GROUP EXECUTED" would have been printed.

A statement group may contain any executable statement; however, a function definition may not occur within a statement group. Either of the statement groups may contain any number of statements. The colon (:) is used to group statements together, and the continuation character (\) allows one statement group to be written over many lines.

```
IF PAGE.BREAK% THEN \
     PRINT FORM.FEED$ :\
     PRINT HEADER$ :\
     PAGE.NO% = PAGE.NO% + 1 :\
     LINE.NO% = 1
```

IF statements may be nested.

```
IF MORE.MASTER THEN \
     IF CURR.REC = M.REC THEN \
          IF MORE.TRANSACTION THEN \
               PRINT PROCESS.TRANSACTION
```

Empty or null statements must be used in some cases to force the proper pairing of the "IF" statement group with the ELSE statement group.

```
IF I < J THEN
     IF A = B THEN \
          IF MORE THEN \
               J + J + 1 \
          ELSE \
               I = I + 1 \
     ELSE \  this else matches second if
ELSE \
     J = J + 1
```

An ELSE will attempt to match the "nearest" IF as shown in this example:

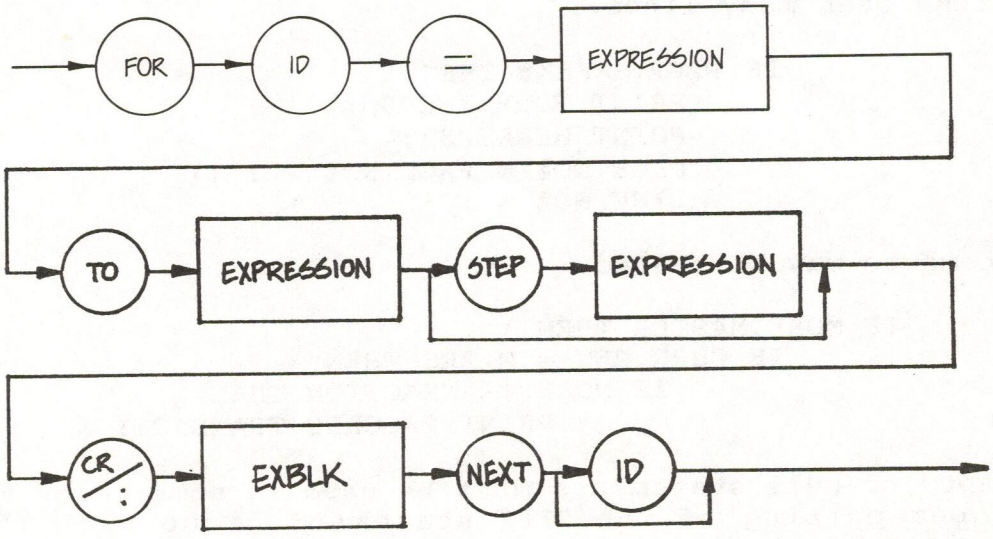
```
IF I < J THEN \
     IF K > L THEN \
          X = 3 \
     ELSE \ this else matches 2nd IF
          Y = 2
```

The following IF statements are invalid.

```
IF A$ THEN GOTO 1Ø     expression must be numeric

IF A < B PRINT X       THEN missing
```

7.3. FOR Loops

FOR loops are one of two looping constructs provided by CB8Ø. A FOR loop consists of a FOR loop header, a statement group, and a NEXT statement. The FOR loop will execute the statements in a statement group zero or more times depending on the values in the FOR loop header.



On each iteration through the loop the index is incremented by the value of the step expression. If the step expression is omitted a value of 1 is used as a default value. The general form of a FOR loop header is shown below.

```
FOR index = <initial exp> TO <final exp> STEP <step exp>
```

The index must be an unsubscripted numeric variable. The type of the FOR loop, either integer or real, is the type of the index. Each of the three expressions are converted to the type of the loop. In other words, if the index of a FOR loop is an integer, the initial, final, and step expressions are converted to integers if any of them are real expressions.

If the FOR loop index is real, any integer expressions will be converted to real values.

```
FOR X = 1 TO J%
```

Since the index X is real the final value J% will be converted to a real value. The step, which in this example defaults to 1, will also be treated as the real constant 1.Ø.

```
FOR I% = 1 TO J%
```

In this FOR LOOP header no conversion is required since the index and final expressions are integers. Programs that use integer indexes and in which the initial, final, and step expressions are integers execute much faster and generate less code than FOR LOOPs with real indexes.

The following pseudo program demonstrates the logic used to execute FOR loops.

```
index = <initial exp>
GOTO loop.end
loop.head:

    [FOR loop statement group]

index = index + <step exp>
loop.end:
     if <step exp> < Ø then \
          if index >= <final exp> then \
               GOTO loop.head \
          else \
     else \
          if index <= <final exp> then \
               GOTO loop.head \
          else

[continue execution with statement following NEXT]
```

As the logic of the pseudo program above shows, loop termination is based on the sign of the step expression. If the step is positive then the loop body is executed while the index is less than or equal to the final expression.

```
FOR I = J TO K STEP 1
   .........

NEXT I
```

The FOR loop statement group above executes K - J + 1 times. If K is greater than J the loop body would not be executed at all.

If the STEP expression is negative, the FOR loop statement group is executed as long as the index is greater than or equal to the final expression.

```
FOR I = -5 TO -1Ø STEP -1
   .........

NEXT I
```

This loop will execute 6 times, with I being assigned values of -5, -6, -7, -8, -9, and -1Ø.

On each iteration of the FOR loop the final and step expressions are reevaluated. The index may be changed within the loop. In addition, the loop may be exited or entered using the GOTO statement.

If the NEXT statement is followed by an identifier, the identifier must be the same as the index of the loop the NEXT statement is terminating. The following FOR loops are equivalent:

```
FOR J = 2 TO K STEP 5          FOR J = 2 TO K STEP 5

NEXT                           NEXT J
```

FOR loops may contain any executable statements including another FOR loop.

```
FOR I% = 1 TO N%
     FOR J% = 1 TO M%
          A(I%,J%) = B(I%,J%) + C(I%,J%)
     NEXT J%
NEXT I%
```

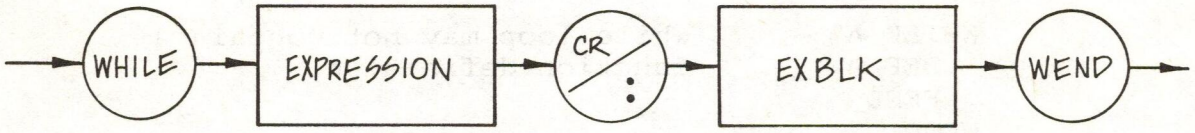
CB8Ø does not limit the depth of nesting of FOR loops. However, in a specific implementation, memory constraints may result in a limit being placed on the number of nested FOR loops. Refer to appendix E for specific limits.

The following FOR LOOPS are invalid:

```
FOR I%(1) = 1 TO N          index must be a simple variable
NEXT I%(1)

FOR J = K TO L STEP M
NEXT K                      NEXT identifier must match index

FOR I = 1 STEP 3            reserved word TO and final value
NEXT                        expression are missing
```

7.4. WHILE Loops

WHILE loops are the second type of looping structure provided by CB8Ø. A WHILE loop consists of a WHILE loop header, a statement group and a WEND statement. The WHILE loop will execute the statements in a statement group zero or more times depending on the value of the WHILE loop header expression.



The expression must be numeric. As with the IF statement, the WHILE loop expression is treated as a logical expression. IF the expression evaluates to zero, the statement following the WEND is executed. If the value of the expression is other than zero, the statements in the statement group are executed.

The expression is evaluated prior to each execution of the statement group.

The following pseudo program demonstrates the logic used to execute WHILE loops.

```
        GOTO loop.end
        loop.head:

                [executable group]

        loop.end:
             if <expression> <> Ø then
                  GOTO loop.head

    [continue execution with statement following WEND]
```

The loop:

```
INTEGER TRUE
TRUE = -1

WHILE TRUE

     ........

WEND
```

will execute indefinitely, since the expression is always true.

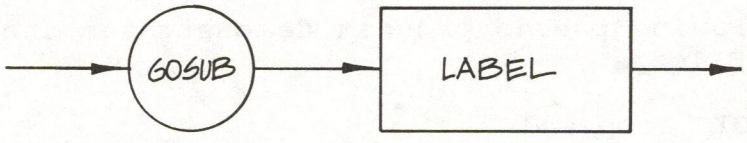
A WHILE loop may be entered by branching to any statement within the statement group, however normal practice is to enter WHILE loops at the loop header.

The following WHILE LOOPS are invalid:

```
WHILE                missing expression
WEND

WHILE A$             expression must be numeric
WEND

WHILE A%             while loop may not contain a
  DEF A              function definition
  FEND
WEND
```

7.5. GOSUB Statements

The GOSUB statement causes statement execution to be transferred to a statement specified by a reference to a label. The address of the statement following the GOSUB statement is saved on a last in first out (LIFO) stack so that statement execution can continue with (or return to) the statement following the GOSUB.



The label must be defined within the program but need not be defined prior to its use in the GOSUB statement. If the GOSUB statement is part of the statement group of a multiple line function then the label must also be part of that statement group. Likewise a GOSUB statement outside of a given function cannot refer to a label within the body of the function.

If the label is not part of an executable statement, the next executable statement after the label is executed.

```
GOSUB 1ØØ

GOSUB PROCESS.ONE.RECORD
```

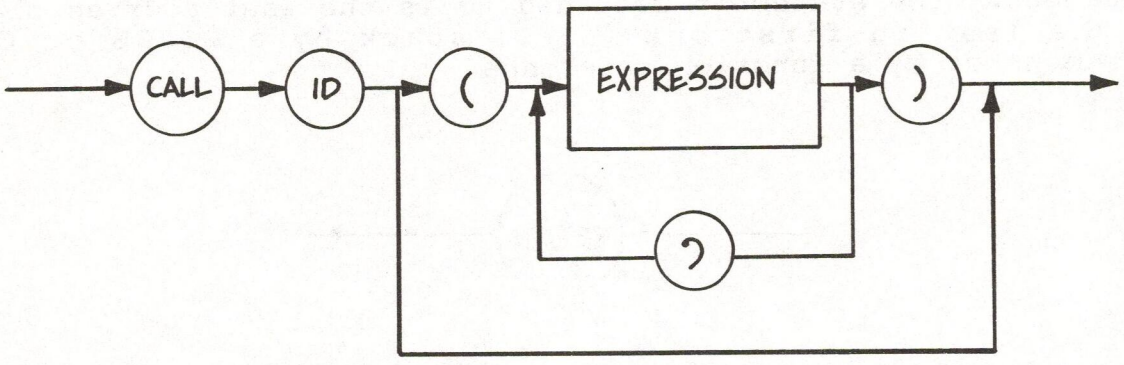
The return statement, described later in this chapter, is used in conjunction with the GOSUB statement.

The following list contains invalid GOSUB statements:

```
GOSUB GET.RECORD:   colon is not in a reference to a label

GOSUB Ø1Ø1B         binary constants cannot be labels

GOSUB NEXT          a reserved word cannot be a label
```

7.6. CALL Statements

CALL statements are used to pass actual parameters to a multiple line function and then execute the function. The address of the statement following the CALL statement is saved on a last in first out (LIFO) stack used by GOSUB statements. This allows execution to be returned to the statement following the CALL. Refer to chapter four for a discussion of functions.



The number of parameters passed by the CALL statement must be the same as the number of formal parameters in the definition of the multiple line function. If the formal parameter is a string, the actual parameter must be a string. Numeric parameters will be converted from integer to real or real to integer as necessary.

```
CALL FN.GET.RECORD

CALL GET.REC(FILE.NM$,REC.NO%,AMOUNT)
```

The following list contains invalid CALL statements:

```
CALL PRINT(REC.NO%)          reserved word cannot be function name

CALL FN.A X,Y                parameters must be enclosed in
                             parentheses

DEF F(A)                     incorrect number of parameters in CALL
  .....
FEND
CALL F(X,Y)

DEF F(A$)                    cannot pass a numeric value to a string
  .....                      formal parameter
FEND
CALL F(X)
```

The multiple line function referenced in a call statement must be defined before it is used in a CALL statement. A function is defined using the DEF statement. Refer to chapter four.

A CALL statement may not be used to call a single line function or to call a program label.

7.7. RETURN Statements

RETURN statements return the program to the statement following the last CALL statement, function reference or GOSUB statement. The statement returned to is the last address placed on the last in first out (LIFO) stack by a GOSUB or CALL statement or by a function reference.

If the RETURN statement is returning from a GOSUB or CALL statement, execution continues with the statement following the GOSUB or CALL, but a value is not passed back.

```
ROUTINE:

    .........

RETURN

GOSUB ROUTINE
X = 3
```

In the example above the GOSUB statement transfers control to the label ROUTINE: and saves the address of the next statement, in this case the assignment to X. After the RETURN statement is executed the assignment is executed.

If the RETURN statement is returning from a function reference the last value assigned to the function name is returned to the expression which referenced the function.

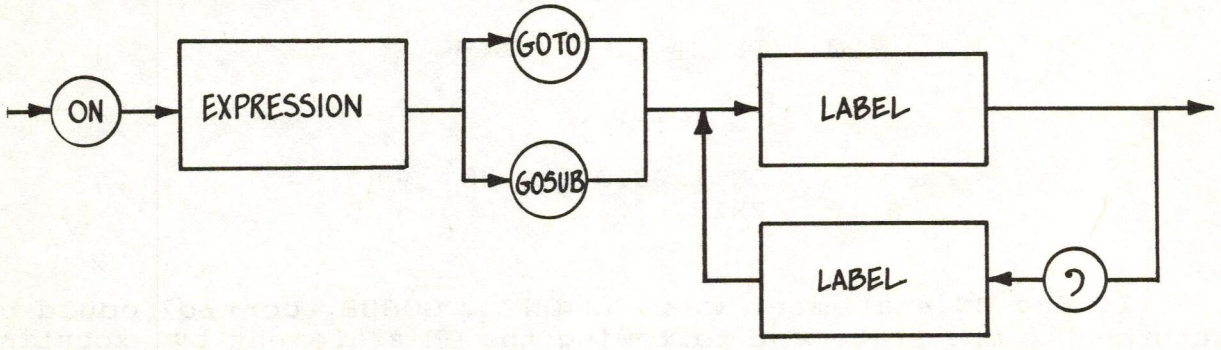
```
DEF ADD.THEM(A,B)
     INTEGER A,B,ADD.THEM

     ADD.THEM = A + B
     RETURN
FEND

X = ADD.THEM(23,56)
```

In this example, the function ADD.THEM returns a value which is assigned to the variable X.

If more RETURN statements are executed then there are addresses on the LIFO stack, the results are undefined. An execution error will not occur.

7.8. ON Statements

ON statements allow execution to transfer to one of a number of labels. Control can be passed using a GOTO statement or a GOSUB statement.



The ON statement is similar to the computed GOTO statement in Fortran. The expression is evaluated and is used as an index to select one of the labels in the list. The expression must be numeric; a real expression will be converted to an integer.

The label references in an ON statement follow the same rules as GOTO and GOSUB statements. Labels need not be defined prior to being used in the ON statement.

The ON statement must have at least one label in the list; there is no limit on the maximum number of labels in an ON statement.

If the expression evaluates to 1, the first label is selected; if it evaluates to 2 the second label is selected, and so forth.

```
I = 3
ON I GOTO LABEL1,LABEL2,LABEL3

LABEL1:
     PRINT 1
     STOP
LABEL2:
     PRINT 2
     STOP
LABEL3:
     PRINT 3
     STOP
```

In the example above the value of I is three so control will be passed to LABEL3 where the PRINT statement will print the number 3. Since the ON statement was an ON ... GOTO, no return value is retained.

The labels in an ON statement need not be defined before they are referenced in the ON statement and they may be in any order in the program. The next example shows an ON statement with one label before the ON statement and one after it.

```
2Ø   PRINT 1

       .......

     ON I GOTO 1Ø, 2Ø
1Ø   PRINT 2
       .......
```

If the ON statement was an ON ... GOSUB, control could be returned to the statement following the ON statement by executing a RETURN statement.

```
I = 2
ON I GOSUB LABEL1,LABEL2,LABEL3
    ........

LABEL1:
     PRINT 1
     RETURN
LABEL2:
     PRINT 2
     RETURN
LABEL3:
     PRINT 3
     RETURN
```

In this example the second label, LABEL2, is selected. When the RETURN statement is executed control is transferred to the STOP statement which is the next statement following the ON ... GOSUB.

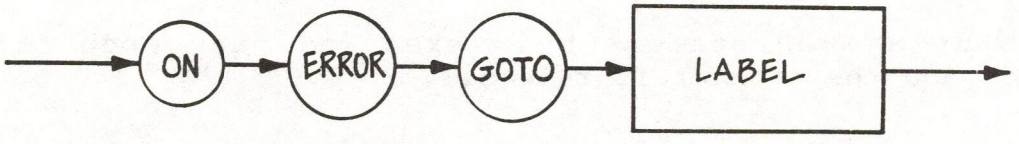
If the index is less than one or greater than the number of labels in the list, the results will be undefined. No execution error will occur.

The following ON statements are invalid:

ON I GOTO 1ØØ 2ØØ	comma missing between labels
ON B$ GOSUB 12, 23	expression must be numeric
ON K-1 1Ø, 2Ø	missing GOTO or GOSUB

7.9. ON ERROR Statements

The ON ERROR Statement is used to trap execution errors allowing the program to process them. The ON ERROR statement is an executable statement which must be executed prior to trapping errors.



When an execution error occurs and the program has executed an ON ERROR statement, execution continues at the first statement following the label referenced in the ON ERROR statement.

```
ON ERROR GOTO PROCESS.ERROR
PROCESS.ERROR:

......
```

In the above example if an error occurs after the ON ERROR statement has been executed, the program will continue execution at PROCESS.ERROR.

When an error occurs, the execution stack is reset. This means that any return address will be lost. For this reason an ON ERROR statement must not be used in the statement group of a multiple line function.

If a program contains multiple ON ERROR statements, the last ON ERROR statement executed will determine the label which it is branched to.

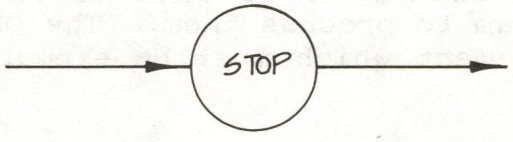
The ON ERROR statement is normally used in conjunction with the ERR and ERRL functions explained in chapter six.

The following list contains invalid ON ERROR statements:

ON ERROR 1ØØ	reserved word GOTO missing
ON ERROR GOSUB ERRQ	GOTO required inplace of GOSUB

7.1Ø. STOP Statements

The STOP statement terminates execution of a program. Control is returned to the operating system.



When a STOP statement is executed, any open files are closed, and the printer is released.

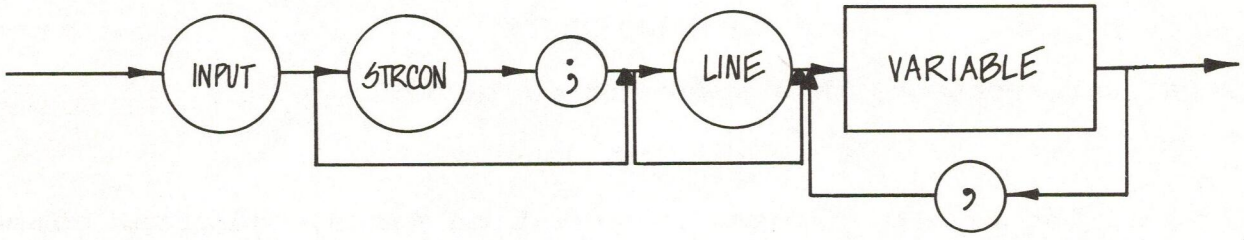
8. INPUT/OUTPUT PROCESSING STATEMENTS

Input and Output processing statements allow data to be transmitted between external devices and CB8Ø variables. This chapter will explain transfer of data to and from the console device and to the line printer. Chapter nine explains input and output between the file system and CB8Ø.

This chapter also explains inputting data from DATA statements to CB8Ø variables. In addition the POKE, RESTORE, and RANDOMIZE statements are explained as well as predefined functions associated will input and output operations.

8.1. INPUT Statements

INPUT statements accept data from the console and assign the data to program variables.



The simplest form of an input statement accepts data from the console and assigns the data to the list of variables.

```
INPUT A, B$, C%
```

The statement above inputs three data items from the console and assigns each data item to a variable in the list. The data input must contain exactly three data fields separated by commas and terminated with a carriage return. A field is a constant followed by a comma or by the end of the input line.

When an INPUT statement is executed, a question mark (?) will be printed on the console followed by one blank space. Then the operator may enter characters in response to the input statement. The response is terminated by a carriage return or when the maximum number of characters allowed has been entered. The maximum will be at least 255 characters. See appendix E for specific implementation limits.

All characters entered in response to an INPUT statement are echoed back to the console. This means that as each character is entered, it is also printed on the console.

Data entered in response to an INPUT statement must contain a field for each variable in the list. In the example above three fields are required. Except for the last field, fields are terminated with a comma. The following input statement requires two fields:

```
INPUT A, B%
```

A proper response for this input statement would be:

```
? 123.45, 45
```

The question mark, and the space which follows it, are printed by CB8Ø. If an incorrect number of fields are entered, a warning message is printed and all the fields must be reentered.

When strings are entered in response to an INPUT statement, the strings may be enclosed in quotation marks. This permits any character except a carriage return to be included in the string. Double quotation marks within the string represent one quotation mark and do not terminate the string.

```
INPUT NAME$
```

A valid response to this statement could be:

```
"Jones, John"
```

If a string is not enclosed in quotation marks, the first comma ends the string. Any other character except a carriage return may appear in a field.

When a field is assigned to a numeric variable the entire field is converted to the internal representation corresponding to the class of the variable. If an unexpected character is encountered in the field, conversion of the field to the internal form is terminated. In other words the number is considered to be the leftmost group of characters that form a valid number.

```
INPUT X
```

The following response to the statement above:

```
? 123.45Q+23
```

would result in X being assigned a value of 123.45. The character "Q" is not expected as part of a number. Thus the remainder of the field is ignored.

When data is entered for assignment to an integer variable, and the magnitude of the integer exceeds the maximum magnitude of CB8Ø integers (32,767) the value assigned to the integer variable is undefined. As with all integer overflow, no error results.

The prompt string is optional in an INPUT statement. If it is present the prompt string is printed in place of the question mark described above. A single blank is still printed prior to accepting input.

```
INPUT "Enter three numbers"; A, B, C
```

This statement prints the following prompt on the console:

```
Enter three numbers
```

Following the prompt one blank is printed and then three fields are accepted as input.

The INPUT LINE statement is a special form of the INPUT statement that accepts one line of input from the console and assigns it to a string variable. The statement:

```
INPUT "What is your name? "; NAME$
```

will accept any characters input until a carriage return is entered. The entire line, excluding the carriage return, is assigned to the string variable NAME$.

Only one variable may appear in an INPUT LINE statement. If only a carriage return is entered in response to an INPUT LINE statement, a null string is assigned to the variable.

The following statements are valid input statements:

```
INPUT "Enter the data"; A,B,C

INPUT LINE X$

INPUT ""; LINE STREET$
```

The following INPUT statements are invalid.

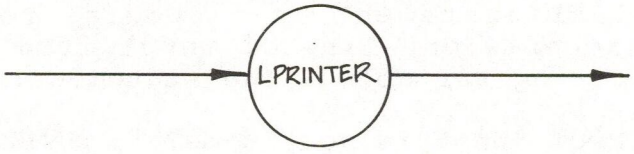
`INPUT LINE A`	must be a string variable
`INPUT "Enter" X`	semicolon missing after prompt
`INPUT A$; C%`	prompt must be a string constant

8.2. CONSOLE and LPRINTER Statements

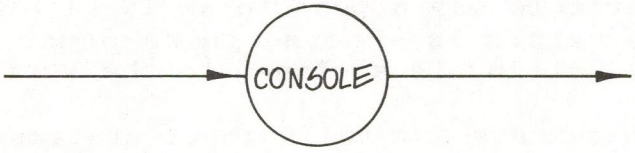
During execution of a CB8Ø program a print control flag is maintained to determine whether output from a PRINT statement is displayed on the list device or on the console. The print control flag is a special variable maintained by CB8Ø; it may not be accessed directly by the programmer. The CONSOLE and LPRINTER statements are used to set and reset this flag. The PRINT statement is explained below.

When the print control flag is reset or false, output from PRINT statements is directed to the console. When the flag is set the output goes to the list device. Initially the flag is reset so the output appears on the console.

The LPRINTER statement sets the print control flag to true so information may be printed on the list device.



The CONSOLE statement resets the print control flag.



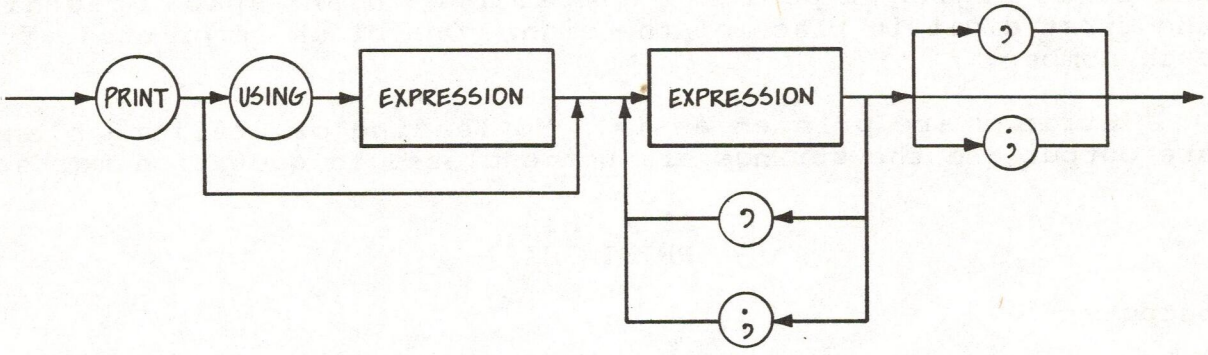
Output resulting from INPUT statement prompt strings is not affected by the print control toggle. It always appears on the console.

The following example uses the LPRINTER and CONSOLE statements.

```
IF LST.REQ THEN \
     LPRINTER   \
ELSE \
     CONSOLE

.............
```

8.3. PRINT Statements

The PRINT statement prints data on the console or line printer depending on whether the print control flag is false or true.



The USING option of the PRINT statement provides formatted output. It is explained in chapter ten. This section will discuss unformatted output.

Each expression in the list is printed on the console or the list device depending on the setting of the print control flag.

```
PRINT X, Y$, I%
```

This statement will print three fields. The first field starts in column one; each of the remaining fields start at the next column after the last number printed that is a multiple of 2Ø. A new line is started after the last field is printed.

The effect of the comma is to force automatic tabbing after the field has been printed. The tab positions are 1, 2Ø, 4Ø etc.

```
PRINT 12,13.78,14
```

prints the following line on the console. The asterisk (*) marks column 1 and the symbol <NL> indicates that a new line is started. This convention representing printed output will be used in this chapter and in chapter nine.

```
*
12                    13.78                  14<NL>
```

Numeric expressions are printed in two formats depending on the value of the number. If the number is greater than or equal to Ø.Ø1 and less than or equal to 99,999,999,999,999 the value is printed in a fixed decimal format.

If the number is outside this range the value is printed in scientific notation with one digit before the decimal point.

```
1.ØE 32

7.218E-1Ø
```

If a number is negative, a minus sign (-) is printed before the first digit. A positive number has a blank space preceding the first digit in place of the sign. One blank is printed after each number.

Strings are printed as is. No leading or trailing blanks are output and the strings are not enclosed in quotation marks.

```
A$ = "HI"
PRINT "HI"
```

outputs:

```
*
HI<NL>
```

If two expressions are separated by a semicolon (;) instead of a comma (,) no automatic tabbing takes place. This means that one field follows directly after the last. Numeric fields will still be separated by a blank since numbers always have a trailing blank appended to them.

```
A = 3
A$ = "HI"
PRINT A;A$;A$
```

outputs:

```
*
3 HIHI<NL>
```

If the last expression in a PRINT statement is followed by a comma (,) or a semicolon (;), a new line is not started.

```
PRINT A+B, B-A, A-B,
```

If the last character is a comma as shown in the example above, the tabbing to the next column which is a multiple of 2Ø occurs but no carriage return is output.

```
PRINT "SAY HI",
```

This statement will output:

```
*
SAY HI_____________
```

The underscore (_) indicates one blank was printed.

The trailing comma or semicolon causes the next PRINT statement to output on the same line as the PRINT statement with the trailing delimeter.

```
PRINT "THIS IS ";
PRINT "A SENTENCE"
```

will output:

```
*
THIS IS A SENTENCE<NL>
```

The next example shows a loop printing a value and automatically tabbing to the next column.

```
FOR I% = 1 TO 3
    PRINT I%,
NEXT I%
PRINT
```

The output from this program is shown below.

```
*
1               2               3                <NL>
```

The following example does not use automatic tabbing.

```
FOR I% = 1 TO 3
    PRINT I%;
NEXT I%
PRINT
```

The output form this program is shown below.

```
*
1 2 3 <NL>
```

The following PRINT statements are invalid:

```
PRINT A+B C+D      missing delimeter (, or ;)

PRINT A +          not a complete expression

PRINT A,,B         missing expression
```

A PRINT statement with no expression list may be used to print blank lines.

```
PRINT
PRINT
```

The two statements above will each start a new line. Thus two blank lines will be printed. The two statements:

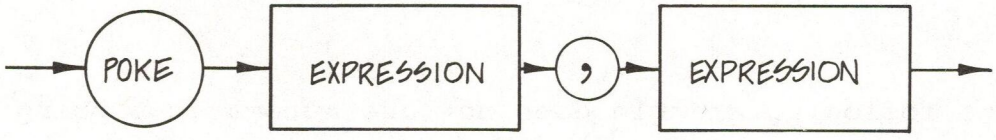
```
PRINT "HI THERE";
PRINT
```

are equivalent to the statement:

```
PRINT "HI THERE"
```

8.4. POKE Statements

The POKE statement places the value of the second numeric expression at an absolute memory location determined by the first numeric expression. The value placed in memory is one byte of data.



The first expression must evaluate to a valid address for the computer being used. However, CB8Ø does not verify that the memory address is valid. The second expression, modulo 256, is placed at this memory location.

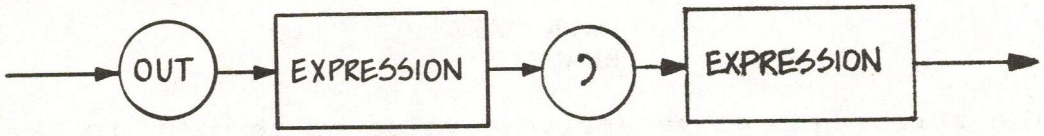
```
POKE MEM.LOC%,VALUE%
```

The absolute addresses assigned to the program code and data area are determined when a module is linked. When using the POKE statement the effect of linking the program must be taken into account.

The expressions must be numeric; if either expression is real, it is converted to an integer.

8.5. OUT Statements

The OUT statement outputs an integer value to a hardware output port. This function is hardware dependent and may not have the same effect on different processors. In addition the OUT statement can also interfere with the operating system being used.

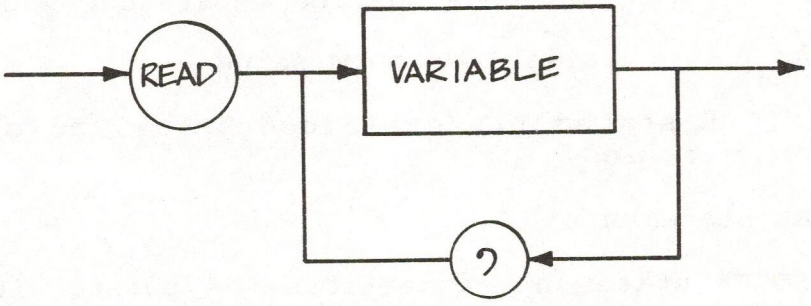



```
OUT PORT%, I%
```

The arguments must be numeric; if either is a real value it will be converted to an integer.

8.6. READ Statements

READ statements accept data defined by DATA statements and assign the values to variables. DATA statements are explained in chapter three.




```
DATA 1Ø, 2Ø, "HI"
READ X, I%, A$
```

The statements above will assign the value 1Ø to the real variable X, an integer value 2Ø to I%, and the string "HI" to the string variable A$.

The following statements are equivalent to the statements above:

```
DATA 1Ø, 2Ø
READ X
READ I%, A$
DATA "HI"
```

Each READ statement assigns the next field in the DATA statement to the variable in the next READ statement. All the DATA statements in a program are treated as one consecutive group of fields. If the variable in the READ statement is numeric the field from the DATA statement is converted into the appropriate

internal representation. When assigning values to variables with READ statements use the same rules as the INPUT statements.

```
DATA "XYZ"
READ I%
```

The statements above assign a value of zero (Ø) to I%. If a READ statement attempts to read a field past the last field in the last DATA statement in the program, an execution error occurs.

```
DATA XYZ
READ A$, B$(I)
```

Executing the statements above will result in an execution error unless there are other DATA statements in the program.

The following READ statements are invalid:

READ A B	comma missing
READ I(");V%	variables must be separated by commas
READ A,,B	variable name missing

The RESTORE statement, explained below, allows the DATA statements to be reused.

8.7. RESTORE Statements

The RESTORE statement repositions the pointer into the data area so that the next value read with a READ statement will be the first item in the first DATA statement in the program.

An example of a RESTORE statement is:

```
RESTORE
```

8.8. RANDOMIZE Statements

The RANDOMIZE statement seeds the pseudo-random number generator so the RND function, explained below, will generate random numbers.

On operating systems that do not provide a time of day function, the seed is generated using the time taken to respond to INPUT statements. If the time of day is available, it is used to generate a random seed.

Thus, on operating systems which do not have the time of day available, it is necessary to execute an INPUT statement prior to using the RANDOMIZE statement. In any event, a RANDOMIZE statement must be used prior to using the RND function in order to generate a different pseudo-random series each time the program is executed.

8.9. Input/Output Predefined Functions

8.9.1. CONSTAT%

The CONSTAT% function returns an integer set equal to the console status. If a character has been entered at the console but not yet read, CONSTAT% returns "true", which is a negative one. Otherwise a false or zero is returned.

```
CONSTAT%
```

8.9.2. CONCHAR%

The CONCHAR% function returns an integer value equal to the next character read from the console. The character read is printed to the console.

The lower eight bits of the returned value are the binary representation of the ASCII character read from the console. The high-order eight bits are always zero.

CONCHAR% always reads one character from the console. If no character has been entered, CONCHAR% will wait until a character has been entered at the console.

```
CONCHAR%
```

For example, if an upper-case letter "A" is entered at the console the CONCHAR% function will return a value of 65.

8.9.3. INKEY

The INKEY function returns an integer set equal to one character read from the console. Unlike the CONCHAR% function, INKEY does not echo the character back to the console.

The lower eight bits of the returned value are the binary representation of the ASCII character read from the console. The high-order eight bits are always zero.

INKEY

The only difference between INKEY and CONCHAR% is that INKEY does not echo the character back to the console. INKEY is useful when control characters or other special characters might be input and the programmer does not desire these characters to be printed. INKEY may also be used to accept passwords.

Some operating systems may require that INKEY be implemented identically to the CONCHAR% function.

8.9.4. INP

The INP function returns an integer equal to the value of the I/O port selected by the argument. This function is hardware dependent and may not have meaning on certain processors. In addition the INP function may interfere with the operating system being used.

The argument must be numeric; if it is a real value it will be converted to an integer.

INP(PORT%)

8.9.5. PEEK

The PEEK function returns an integer equal to the value of the memory location selected by the parameter. The memory location must be valid for the computer being used. However, CB8Ø does not check on the validity of the memory address.

The parameter must be numeric; if it is real it will be converted to an integer.

PEEK(MEM.LOC%)

8.9.6. POS

The POS function returns the current position in the output line. The value returned is an integer.

POS returns the number of characters plus one that has been output to the console or list device since the last carriage return. In other words, POS returns the next position in which a character will be printed.

Output to the console may be generated by PRINT statements or by INPUT statement prompts.

```
POS
```

The following statements:

```
PRINT
PRINT POS,POS
```

will print the numbers 1 and 2Ø.

8.9.7. RND

The RND function returns a real value which is a uniformly distributed pseudo-random number between Ø and 1.

```
RND
```

8.9.8. TAB

The TAB function prints blank characters until the value POS would return is equal to the argument. If the value of the argument is less than or equal to the current position to be printed, a new line is started and then the TAB function is executed.

The argument must be numeric; if it is a real value it will be converted to an integer. A zero or negative argument will cause an execution error.

```
TAB(I%)
```

If the console cursor position has been changed using special control characters, or if the position has been changed using the CONCHAR% function (explained previously), the TAB function will not provide the desired results.

The TAB function may only be used in PRINT statements.

9. FILE PROCESSING STATEMENTS

A file is a collection of data items stored on an external device other than the console or line printer. CB8Ø is not concerned with the physical storage of the data but rather with the logical organization of the data.

This chapter will explain the statements that open or create files, access files, and close or delete files. In addition, predefined functions which involve file accessing will be explained.

9.1. File Description

CB8Ø supports two types of files: stream and fixed. In a stream file information is placed in the file as a stream of fields with no record structure. The file is a continuous stream of individual data items; there is no implied relationship between data items.

Each field in a stream file is separated from the next field with a comma or by new line characters. With most operating systems the new line characters are a carriage return and line feed pair.

Fixed files also have fields of data separated by commas. However, the fields are grouped into fixed length records. Unused space in records is padded with blanks and the record is terminated with new line characters.

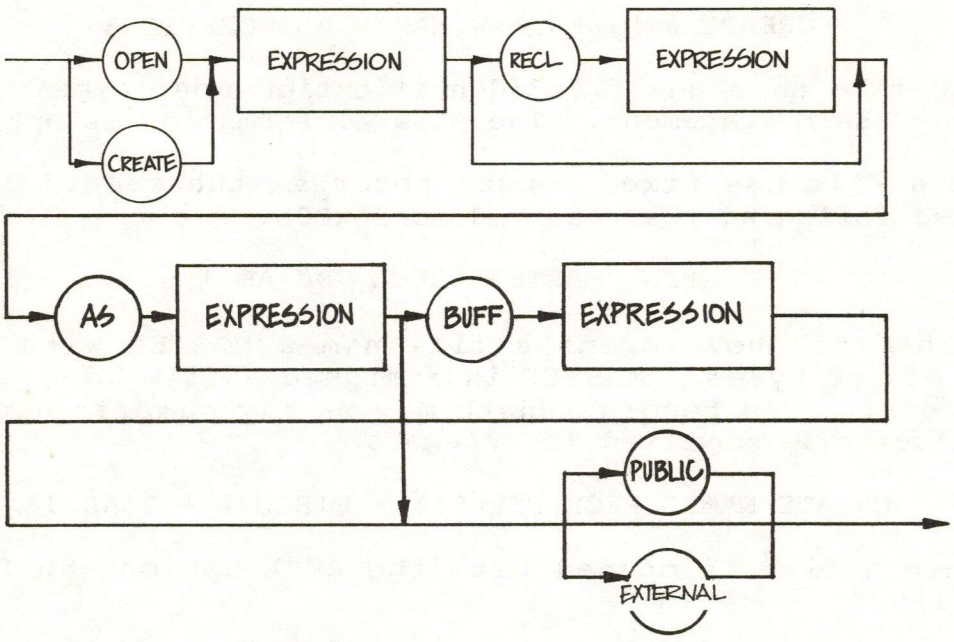
In fixed files the new line characters are part of the record. Thus the minimum record size is two bytes. Of course no information could be stored in a file with a record length of two! The only limit on the maximum record length is that it must be expressed as an integer value.

In general, CB8Ø files are made up of ASCII characters. This allows the file to be conveniently displayed on the system display using operating system utilities. Binary files can be built and accessed with certain restrictions explained in this chapter.

9.2. OPEN and CREATE Statements

Before data items can be written to a file or read from a file an interface must be established between CB8Ø and the operating system. Characteristics such as the device selection, file names, and buffer requirements must be defined.

CB8Ø provides two statements to define files, the OPEN and CREATE statements. The OPEN statement is used for accessing existing files; the CREATE statement creates a new file with no data in it.



The first expression in an OPEN or CREATE statement is a string expression which evaluates to a valid file name for the operating system being used. A particular operating system may restrict the characters in the file name and the length of the name.

The expression following the reserved word AS assigns a CB8Ø file identification number to the file. All future references to the file will use this number.

The file identification number may be any numeric expression. If the expression evaluates to a real value it will be converted to an integer. An execution error will occur if the value is zero or negative, or if it is greater than the maximum

number of files which may be open at one time. See appendix E for current limits. If the file identification number in an OPEN or CREATE statement is currently assigned to another file an execution error will occur.

A file is considered open when it has been assigned a file identification number by an OPEN or CREATE statement. The same number may not be used for two open files. The number of files that CB8Ø allows to be open at one time is dependent on the implementation. See appendix E for current limits. Some operating systems may impose further restrictions on the number of files that may be open at one time.

```
OPEN "TEST" AS 4

CREATE W.DISK$ + W.NAME$ AS WORKFILE%
```

The file name and file identification must appear in every OPEN and CREATE statement. The other information is optional.

If a file has fixed length records, the record length is specified following the reserved word RECL.

```
OPEN "MASTER" RECL 7ØØ AS 1
```

The statement above opens a file named MASTER with a record length of 7ØØ bytes. MASTER is assigned a file identification number of 1. The record length may be any numeric expression. Real values are converted to integers.

```
CREATE NAME$ RECL FIELD1% + FIELD2% + 2 AS J%
```

When a file is opened with the RECL option the file is a fixed file.

The reserved word BUFF is used to specify the number of internal buffers to maintain for the file. If no buffers are specified, a value of one is assumed. The size of a buffer depends on the implementation, but is normally chosen so that it is the amount of data that can be accessed by one call to the operating system. See appendix E for current specifications.

```
OPEN A$ AS 4 BUFF 1Ø
```

The statement above opens a file with 1Ø buffers assigned for its use. Multiple buffers are always stored consecutively in memory. The MFRE function can be used to determine the amount of available memory prior to determining the number of buffers to use.

The BUFF option may not specify more than one buffer when the file will be accessed randomly. Random access is explained later in this chapter.

With CB8Ø the amount of buffer space required is independent of the record length. There is no requirement that the complete record be held in memory at one time. CB8Ø provides all the deblocking necessary to support large records in a system with limited memory.

The following OPEN and CREATE statements are invalid:

OPEN "TEST"	file identification missing
CREATE A$ AS 1 RECL 1ØØ	reserved words in wrong order
CREATE 3 AS 2	file name must be of type string
OPEN FN$ BUFF 1Ø AS 2	reserved words in wrong order

The INITIALIZE statement resets the operating system after diskettes or other storage media have been replaced. This prevents the operating system from writing data to the wrong place on the storage media.

When changing removable media the INITIALIZE statement must not be executed until the swapping is complete and the devices on which the media is placed are ready.

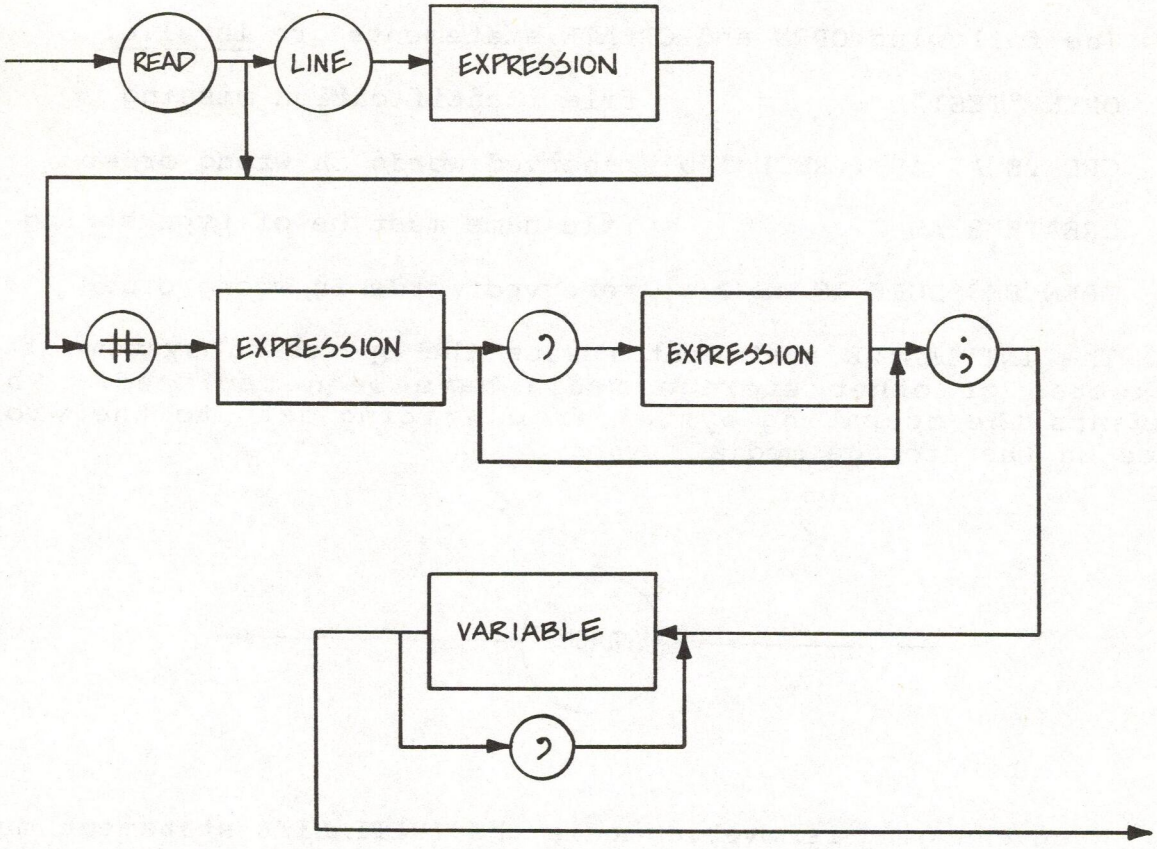
9.3. File Accessing Methods

Files may be accessed in three ways: sequentially, randomly, or byte at a time. These methods may be used interchangeably with the provisions that only fixed files may be accessed randomly.

This section explains the file READ, file PRINT, and PUT statements.

9.3.1. Reading Files

Files can be read sequentially and randomly using the file READ statement.



A file READ statement specifies the file identification number for the file to be read, and a list of variables to which data items read from the file are to be assigned. An optional record number may be specified to select the record to be read. This may only be used with fixed files.

The file identification number and optional record number may be any numeric expression. If either expression is real it will be converted to an integer value. An execution error will occur if the file identification number does not evaluate to an integer assigned to an open file.

```
READ # 1; A, B$, C%
```

This statement reads the next sequential record from a file with a file identification number of 1 and assigns the first three fields to the variables A, B$, and C%. In the case of variables A and C% the fields are interpreted as numbers and converted to the internal format for real and integer variables respectfully.

There is a fundamental difference in the way fixed record length files and stream files are treated when reading the files. If a file has fixed records, any fields that were not read by the READ statement are skipped. This means the next sequential read will read a new record even if fields were left unread in the previous record.

If a file is a stream file, one field is read after another and no logical organization is assumed. Consider a file with the following records:

```
1,2,3,4CRLF

5,6,7,8CRLF

9,1Ø,11,12CRLF
```

This file is accessed with the following READ statements:

```
READ # 1; A,B,C

READ # 1; D

READ # 1; E,F
```

If the file is a stream file, that is no record length was specified when the file was opened, the variables A through F would be assigned the following values:

```
A = 1   B = 2   C = 3
        D = 4
  E = 5   F = 6
```

However, if the file was opened with a record length specified the variables would be assigned the following values:

```
A = 1   B = 2   C = 3
        D = 5
  E = 9   F = 1Ø
```

If the records were fixed length records they would have been padded with blanks. This is not shown above.

Another difference between reading fixed length files and stream files occurs when a carriage return is encountered as a field delimeter. If the file is fixed and an attempt is made to read past a carriage return, an execution error occurs. When reading a stream file, a carriage return is treated just like a comma. Thus, when reading a fixed file, one READ statement reads one record assigning the fields in the record to variables in the variable list.

A READ statement can select a specific record to read

instead of reading the next sequential record. The file being read must be a fixed record length file. This type of access is called random access.

```
READ # 1, 12; A, B, C(I,J)
```

The statement above will read the twelfth record from file 1. The first three fields in the record will be assigned to the variables A, B, and C. If a record in this file has less than three fields or the file was a stream file, an execution error will occur.

The first record in a file is record one. An execution error occurs if the record number is zero. The record number is treated as an unsigned sixteen bit integer. This means that "negative" record numbers can be used for record numbers greater than 32767.

If an attempt is made to read a file past the last record in the file, CB8Ø will report that end of file has been reached. The section in this chapter on file exception processing explains how the programmer can process an end of file condition. An end of file exception also occurs when a random read attempts to read a record that does not exist.

Sometimes it is necessary to position to a specific record in a file and then read the file sequentially.

```
READ # 1, 7;
```

The statement above will position file 1 to the beginning of record 7. No data is read from the file. An execution error will occur if the file is a stream file.

The READ LINE statement is similar to the INPUT LINE statement explained in chapter eight. It will read one complete line of data from a file and assign the information read to a string variable.

```
READ #FILE.NO%; LINE D$
```

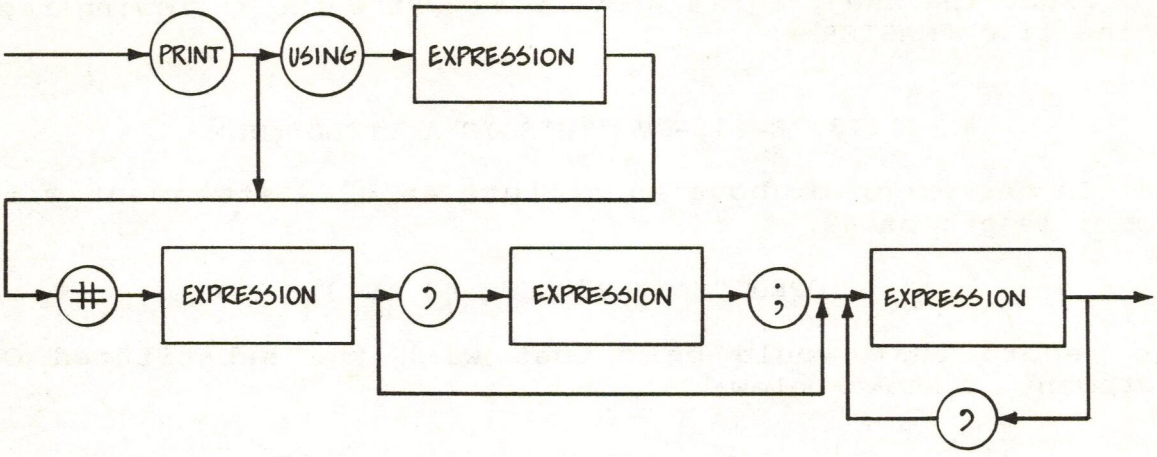
This statement reads the next sequential record from the selected file and assigns the entire record up to but not including the new line characters to the string variable D$. Only one variable may be used after the reserved word LINE and it must be a string variable.

The READ LINE statement can also read a random record.

```
READ #F%, R%; LINE X$
```

9.3.2. Writing to Files

Files can be written to either sequentially or randomly using the file PRINT statement. This section describes unformatted output to files. Chapter 1Ø explains formatted output.



A file PRINT statement specifies the file identification number for the file begin printed to, and a list of expressions which are evaluated and output to the file. An optional record number may be specified to select the record to output. This may only be used with fixed files.

The file identification number and optional record number may be any numeric expression. If either expression is real it will be converted to an integer value. An execution error will occur if the file identification number does not evaluate to an integer assigned to an open file.

```
PRINT # 1; A%, B, C$
```

This statement prints three fields to the next sequential record in the file with a file identification number of 1. The first two fields are separated by commas and the last field is followed by new line characters.

When a string is output to a file it is enclosed in quotation marks. Numbers are output to a file following the same formatting rules used for output to the display.

If the file is a fixed file, sufficient blank characters are output after the last field and before the new line characters to ensure each record is the length that was specified when the file was opened. If the data output to the file results in a record length that exceeds the fixed record length, an execution error occurs.

```
OPEN "MASTER" AS 3
X = 21.73
Y = .ØØØØ7
I% = -72
A$ = "THIS IS A FIELD"
PRINT # 3; X, Y, I%, A$
```

Executing the program above will write the following record to the file "MASTER":

```
*
21.73,7E-Ø4,-72,"THIS IS A FIELD"<NL>
```

In the program above substitute an OPEN statement with a record length of 4Ø.

```
OPEN "MASTER" RECL 4Ø AS 3
```

The record that would be output with the substituted OPEN statement is shown below:

```
*
21.73,7E-Ø4,-72,"THIS IS A FIELD"____<NL>
```

An execution error would occur if the record length had been less than 34.

A file PRINT statement can select a specific record in a file to direct output to. This type of access is called random access. To use random access the file must be a fixed record length file.

```
PRINT #3, 4; C(I), A$+B$
```

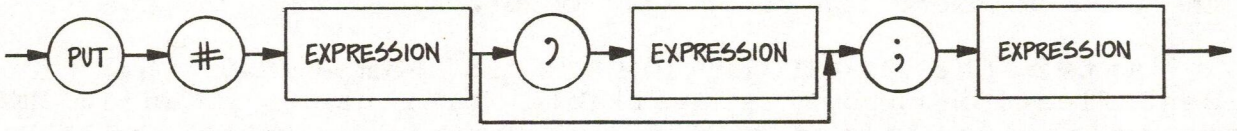
The statement above will output record four to the file using three as a file identification number. The record contains two fields.

If an attempt is made to output to a file and insufficient space is available on the file system, a file exception will occur. The section in this chapter on file exception processing explains how the programmer can trap this condition.

The following file PRINT statements are invalid:

```
PRINT 3; A          missing pound sign

PRINT #I, J;        missing expression list

PRINT # 2; A+1; B   commas must separate expressions
```

The PUT statement will write one byte to the selected file. The byte may be any value between Ø and 255.



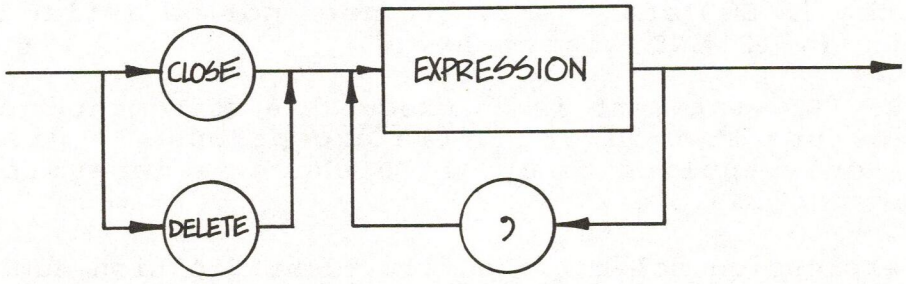
Both expressions must be numeric; if one of them is a real expression, it will be converted to an integer.

The PUT statement allows binary data to be written to a file. No delimiters or other characters are added to the data output.

```
PUT 3, I%
```

9.4. Terminating Access To Files

CB8Ø provides two statements which terminate access to files, the CLOSE and the DELETE statement. To use these statements the file must be open.



The CLOSE statement tells the operating system that no further access to the file is required. Any interfaces established by the OPEN or CREATE statement are terminated. All information in the file is retained.

```
CLOSE 3

CLOSE TEMP1%, TEMP2%
```

The DELETE statement instructs the operating system to remove the file from the system directory. No information is retained about the file.

```
DELETE 1

DELETE INDEX% + 3
```

The expressions in CLOSE or DELETE statements must be numeric; if they evaluate to real values, they will be converted to integers. Each expression must evaluate to a valid file identification number and that number must refer to an open file. Otherwise an execution error will occur.

After the CLOSE or DELETE is complete, the file identification number is available for reuse. If an IF END statement, explained below, is associated with the file identification number the association is terminated.

9.5. File Exception Processing

The IF END statement traps file system exceptions and allows the programmer to take appropriate action.

The label reference must refer to a label defined within the scope of the IF END statement. It need not be defined prior to its use in an IF END statement.

The IF END statement is an executable statement. It must be executed before it will trap file exceptions. A given IF END statement only applies to one file which is determined by the expression.

The expression selects the file identification number of the file for which exception processing is desired. The expression must be numeric; a real expression will be converted to an integer.

```
IF END # 1 THEN 2ØØ

IF END # WORK.1% THEN FILE.EOF
```

The following types of exceptions are trapped by the IF END statement:

(1) READ PAST END OF FILE

(2) DISK OR DIRECTORY FULL DURING PRINTING TO A FILE

(3) ATTEMPT TO OPEN A FILE THAT DOES NOT EXIST

If any of these exceptions occur, the file processing system determines if an IF END statement has been executed for the file identification number of the file on which the exception occurred. If no IF END statement is in effect, an execution error occurs. Otherwise execution continues at the statement with the label referenced in the IF END statement.

```
IF END # 3 THEN 2ØØ
.......

2ØØ REM PROCESS EXCEPTION FILE 3
```

Prior to transferring control to the exception processing routine, the execution stack is adjusted so that all return addresses saved on the LIFO stack are retained.

The following IF END statements are invalid:

```
IF END 7 THEN 2ØØ          missing #

IF END # 7 2ØØ             missing THEN

IF END # I% THEN PEOF:     label reference may not have colon
```

A program may have any number of IF END statements for the same file identification number. The most recently executed IF END for a given identification number is the IF END statement in effect when an exception occurs.

An IF END statement may use a file identification number that is not currently being used by an open file. This allows the IF END to trap exceptions when an OPEN statement is executed.

9.6. File Predefined Functions

9.6.1. GET

The GET function accepts one byte of data from the file selected by the parameter. The parameter must be numeric; if it is real it will be converted to an integer.

```
GET(FILE.ID%)
```

The value returned by the GET function is an integer. It may be any value between Ø and 255. In other words, GET returns binary data from a file.

9.6.2. LOCK

The LOCK function locks a record in the file selected. This prevents other programs from updating that record. LOCK returns an integer value which is true (-1) if the record could be locked, and a false (Ø) if it could not be locked.

```
LOCK(FILE.ID%,REC%)
```

Both arguments must be numeric; if either evaluates to a real value, it is converted to an integer.

If the operating system being used does not support record locking, no action occurs and a value of true is returned by LOCK.

9.6.3. RENAME

The RENAME function renames a file. The file being renamed must not be open. The arguments must both be string expressions. The value returned by RENAME is an integer value which is true (-1) if the rename was successful and false (Ø) if the new name for the file already exists.

```
RENAME(NEW$,OLD$)
```

An execution error occurs if the old file name does not exist.

9.6.4. SIZE

The SIZE function returns the size of the file specified by the parameter. The value returned is an integer equal to the number of 1Ø24 byte blocks contained in the file. In other words, the SIZE function returns the size of the file, in bytes, divided by 1Ø24.

The argument must evaluate to a string. The string represents the file name.

```
SIZE(FILE$)
```

The file does not need to be open.

```
SIZE("NAME")

SIZE(TEMP1$ + ".$$$")
```

Some operating systems support wild card selections for files. For instance CP/M allows the asterisk (*) and question mark (?) to represent matches with a variety of characters. The asterisk matches any name or type extension while the question mark matches any one character in the name or type.

For example "*.BAS" refers to all files with a type extension of BAS. The SIZE function accepts wild card specifications when operating systems such as CP/M support this feature.

```
SIZE("*.TMP")

SIZE("CB8Ø.OV?")
```

If the file contains no data or if the file does not exist, SIZE returns a zero.

9.6.5. UNLOCK

The UNLOCK function performs the opposite action as the LOCK function. The parameters evaluate to a file identification number and a record number. UNLOCK attempts to unlock the selected record. If, after executing the LOCK function, the record is unlocked, a true (-1) is returned. Otherwise a false (Ø) is returned.

```
UNLOCK(FILE.ID%,REC%)
```

Both arguments must be numeric; if either evaluates to a real value, it is converted to an integer.

If the operating system being used does not support record locking, no action occurs and a value of true is returned.

If the record was already unlocked when the UNLOCK function was executed, a true is returned.

1Ø. FORMATTED OUTPUT

CB8Ø allows output generated by a PRINT statement to be formatted under the control of a using format string. This form of a PRINT statement is called a PRINT USING statement. It may be used with output and may be directed to a disk file, the console or the line printer.

Formatted output to files is explained in the last section of this chapter. Output to the console or line printer is controlled by the LPRINTER and CONSOLE statements.

1Ø.1. Using Strings

A print statement that has the reserved word USING followed by an expression and a semicolon is a PRINT USING statement. The syntax diagrams in chapters eight and nine showed this form of the PRINT statement.

The using string must be a string expression. It consists of literal characters, numeric fields, and string fields. The following example shows using strings:

```
PRINT USING A$; A,B,C

PRINT USING USING.STRING$(3); #1, REC%; A$,B$

PRINT USING "The amount owed is $$#,###,###.##"; BALANCE
```

When a PRINT USING statement is executed, the next expression in the expression list is evaluated. The using string is then scanned. Literal characters are output as they are encountered. When a field is located that matches the type of the expression, the expression is output in the format dictated by the format field.

No delimiters, automatic spacing or other characters are output except at the end of the print statement a new line is started unless the expression list ends in a comma or semicolon. In the case of a disk file with fixed length records the record is padded with blanks if necessary prior to outputting the carriage return and linefeed.

If the expression list contains a string expression there must be at least one string field in the using string otherwise an execution error will occur. Likewise, there must be a numeric

field in the using string if there is a numeric expression in the expression list.

The table below lists the format field characters supported by CB8Ø. The numeric and string fields consist of combinations of these characters. The backslash acts as an escape character to force the next character to be treated as a literal character instead of a field character. This does not cause a conflict with continuation characters since a backslash character within a string constant is treated as a character in the string. For example:

```
PRINT USING "The part is \# #####"; \
                        MASTER.PART.NUMBER%
```

FIELD CHAR	FUNCTION
#	Digit position in a numeric field
$$	Float a dollar sign in a numeric field
**	Asterisk fill a numeric field
-	Leading or trailing sign in a numeric field
,	Place commas every third digit before decimal point in a numeric field
.	Decimal point position in a numeric field
^^^^	Exponent position in a numeric field
&	Variable length string field
/..../	Fixed length string field
!	Single character string field
\	Escape character. Treat next character as a literal

1Ø.2. Numeric Fields

A pound sign (#) indicates one numeric position. For example the following statement:

```
PRINT USING "###"; I%
```

defines a field of three positions in which to print I%. If I% is set equal to 3 then the result would be printed as:

```
__3
```

The underscore (_) will be used in this chapter to indicate a blank is printed in the space. In this example two blanks and then the numeral three is printed. The value is right justified in the field and filled with leading blanks.

The following table lists the results which will occur with other values of I%. The using string remains "###".

I%	result
1Ø	_1Ø
999	999
-1Ø	-1Ø
1ØØØ	%1ØØØ
-999	%-999

The last two examples show numbers that do not fit into the field. In these cases the overflow is indicated by printing a percent sign (%) followed by the number in the print format that would be used with printing without a using string. Another example of field overflow is shown below.

```
PRINT USING "###"; 1ØE1Ø
```

The output from this statement would be:

```
%1.ØE 11
```

One decimal point may appear in a numeric field. The following table shows examples using a decimal point:

value	field	result
1Ø.1Ø	##.##	1Ø.1Ø
1ØØ.789	####.##	_1ØØ.79
945.673	####.##	_945.67

Observe that values are rounded to fit the field to the right of the decimal point. Also, if no digits exist before the decimal point and there are one or more digit positions in the format string before the decimal point, a leading zero is printed.

value	field	result
Ø.78	.###	.78Ø
Ø.78	#.#	Ø.8
Ø.999	#	1

If one or more commas appear in the numeric field, the results will be printed with commas inserted every third digit before the decimal point. Each comma in the numeric field serves as a digit position specifier and each comma that actually is printed uses one of the available digit positions. The following table shows the use of the comma in numeric fields.

value	field	result
1ØØØ.Ø	#,###.##	1,ØØØ.Ø
1ØØ.Ø	#,###.##	__1ØØ.Ø
7654321	##,###,###	_7,654,321
7654321	#,#######	7,654,321
7654321	#,######	%7654321

The commas do not have to be placed where they will occur in the output and only one comma need be used to cause all the necessary commas to be printed. However, the total number of positions available in the field is determined by the number of pound signs and the number of commas.

Numeric expressions may be printed in an exponential format by appending one or more uparrows (^) to the end of the numeric format field. The exponent will always use four positions when it is printed. From one to four uparrows may be used to specify the exponent.

value	field	result
1ØØ	##^^^^	1ØE_Ø1
-7751.21	##.##^	-7.75E_Ø3
.ØØ1234	###^^^^	123E-Ø5
Ø	##^^^	ØE_ØØ

Commas will not be printed in a numeric field with an exponent. If commas occur in the field, they are treated as pound signs.

value	field	result
123456	#,###^^	12346E_Ø1
234	#,###^^^^	23400E-Ø2

In the example above the numeric field has five positions for the digits. This requires that the number be rounded to five significant digits.

In the examples above blank characters were placed in any leading field positions that were unused. Instead of the blanks an asterisk (*) may be used as a fill character by placing two asterisks at the beginning of a numeric field.

value	field	result
754	**###	**754
-21	**###	**-21
12345	**###	12345

The two asterisks are counted as two numeric positions just as pound signs are. Only if blanks would normally be used to fill the field are the asterisks printed in the place of blanks.

Asterisk fill may not be used in fields with an exponent format. A single asterisk is treated as a print character and not as part of a numeric field.

A dollar sign ($) may be printed to the left of the first digit in a numeric field. This is referred to as floating a dollar sign and is specified by placing two dollar signs at the beginning of a numeric field.

value	field	result
1Ø.1Ø	$$###.##	__$1Ø.1Ø
1ØØØ.ØØ	$$###.##	$1ØØØ.ØØ
1ØØØ.ØØ	$$#,##.##	$1,ØØØ.ØØ
1ØØØØ.ØØ	$$#,##.##	1Ø,ØØØ.ØØ

Blanks are used to fill the field when a floating dollar sign is a part of the numeric field. As with the asterisk fill, the two dollar signs are counted as two numeric positions. The last example above shows the dollar sign is only printed if a position is available.

Floating dollar signs may not be used in fields with an exponent format. Also, if the numeric expression output into the field is negative, the minus sign is printed in place of the dollar sign.

value	field	result
-1Ø	$$##.##	_-1Ø.ØØ
1Ø	$$##.##	_$1Ø.ØØ

A single dollar sign is treated as a print character and not as part of a numeric field.

Normally a negative number has the sign floated to the left of the first digit in the number being printed. By placing a minus sign (-) as the first or last character of a using string the minus sign may be placed in a fixed position in the field.

value	field	result
-123.456	###.###-	123.456-
-123.456	-###.###	-123.456
-12.345	-###.###	-_12.345
Ø.3456	##.###-	__.347_
1ØØ.ØØØ1	-###.##	_1ØØ.ØØ

As the examples above show, if the sign of the expression is positive a blank is printed in place of a sign.

1Ø.3. String Fields

There are three types of string fields: single character, variable length, and fixed length. A single character field is specified by an exclamation mark (!). It prints the first character of a string expression.

```
PRINT USING "!"; "ABC"
```

will print the letter A. Successive exclamation marks will print the first letter of successive string expressions. In other words, each exclamation mark is a separate string field.

```
PRINT USING "!! !"; "XY","UV","PQ"
```

will output:

```
*
XU_P<NL>
```

This is the same notation used in chapter eight. The asterisk (*) marks column 1 and the <NL> indicates that a newline is started.

A single ampersand (&) is used to represent a variable length string field. The ampersand results in the entire string being printed with no editing.

```
PRINT USING "&"; "THIS IS A STRING"
```

would print:

```
*
THIS IS A STRING<NL>
```

The next example uses both variable length and single character string fields.

```
PRINT USING "& !. &";"Jim", "Allen","Smith"
```

will print:

```
*
Jim A. Smith
```

The third type of string field is the fixed length field. This field is delimited by slashes (/). The size of the field is the number of spaces or characters between the slashes plus two. Each slash is one position in the fixed field and each character between the slashes is also counted in the size of the field.

```
PRINT USING "/   /"; "HI THERE"
```

will output:

```
*
HI TH<NL>
```

The string field in the example above consists of three spaces and the two slashes. Thus the field has a total length of five characters. The left five characters of the string expression are printed.

Any characters may be placed between the slashes. These characters are ignored but may be used to document the use or size of the field. The examples below demonstrate this:

```
PRINT USING "/ NAME /"; NAME$

PRINT USING "/...5...9/";A$ + B$
```

If the string expression evaluates to a string shorter than a fixed length field, the expression is left justified in the

field. Blanks are used to fill the field on the right.

```
PRINT USING "/...5...9/";"XYZ"
```

will output:

```
*
XYZ_______<NL>
```

Both string and numeric fields may be mixed in a using string.

```
PRINT USING "#.# XYZ &"; 7.2, "ABC"
```

The output from this statement is shown below:

```
*
7.2 XYZ ABC<NL>
```

The characters XYZ and the space before and after them are literal characters. They appear in the output just as they are in the using string.

A using string is reused if the end of the using string is reached and there are still more expressions from the expression list to be printed. The using string is reused by wrapping around to the beginning of the string.

```
PRINT USING "!"; "A","B","C"
```

will output:

```
*
ABC<NL>
```

The using string was reused three times to allow each expression to be printed. In the following example each field in the using string will be used once and then the first field will be used a second time.

```
PRINT USING "## X &"; 5,"HI",6
```

will output:

```
*
_5_X_HI_6_X_<NL>
```

After the three fields were output a trailing "_X_" was printed. As each expression is printed, including the last expression, any literal characters following the field in the using string are output. As soon as a string or numeric field is encountered no more characters are printed. Also, if the end of the using string is reached, the using string will not be reused

just to print literal characters.

```
PRINT USING "THIS IS A NUMBER ## TO PRINT"; 99
```

The output from this statement is:

```
THIS IS A NUMBER 99 TO PRINT<NL>
```

It is possible for characters in a string or numeric field to be treated as literal characters.

```
PRINT USING "&## X &";29
```

will output:

```
*
&29_X_<NL>
```

The expression is numeric. Thus, every character in the using string is treated as a literal character until a numeric field is found. In this case the ampersand is printed as a literal character. After the last expression has been printed (in this example there is only one expression) all characters in the using string are printed as literal characters until the next string or numeric field is found. This results in the "_X_" being printed but not the second ampersand.

The following PRINT USING statements are invalid:

```
PRINT USING "###"; A$     no string field but string expression

PRINT USING "/  "; B$     missing closing slash

PRINT USING "##" X+Y      missing semicolon
```

1Ø.4. Escape Characters

The backslash (\) serves as an escape character to force the next character to be a literal character. This allows characters such as pound signs (#) and ampersands (&) to be treated as literal characters.

```
PRINT USING "\###"; 1Ø
```

The output from this statement is:

```
*
#1Ø<NL>
```

The backslash causes the first pound sign to be treated as a literal character. If the backslash is the last character in a using string an execution error occurs.

A backslash may be printed as a literal character by placing two backslashes in the using string.

```
PRINT USING "\\#";3
```

will output:

```
*
\3<NL>
```

1Ø.5. Print Using to Files

The PRINT USING statement also can be used to write formatted data to files. The same using strings explained in this chapter may be used with file PRINT statements.

```
PRINT USING "&"; #1; A$
```

This statement outputs one record to the selected file. The record is terminated with new line characters. Quotation marks are not placed around string data and fields are not delimited by commas.

```
PRINT USING A$+B$; F1%,REC%; X,Y,Z
```

The statement above shows output to a file using random access.

If the file being output to is a fixed file, the record will be padded by blanks as required to ensure that it is the proper length.

11. PROGRAM MODULES

In the previous chapters programs have been presented as single modules with all variables and procedures being known only to that program. This chapter will explain how multiple modules may be linked together into one program allowing procedures and variables in one module to be accessed by the other module.

In addition it is sometimes necessary to divide program tasks into logical groupings because if all the desired functions were implemented in one program it would not fit into memory. This chapter will explain how programs may be overlayed or replaced by other programs and still pass information among themselves.

11.1. Public and External Functions

Multiple line functions, explained in chapter four, may be compiled separately, forming a module. This module may be linked with another CB8Ø module or a module created by a relocatable assembler such as RMAC.

A very important point must be emphasized when combining modules to form a program. Only one of the modules may contain executable statements in its executable group. The other modules must only contain multiple line functions.

A function that can be referenced in another module is called a public function.

```
DEF THIS.IS.A.FUNCTION PUBLIC
    INTEGER THIS.IS.A.FUNCTION

    .......

FEND
```

THIS.IS.A.FUNCTION is a public function. If a module contains this function, and it is linked with another module, the second module may reference THIS.IS.A.FUNCTION. The following program could access function THIS.IS.A.FUNCTION:

```
DEF THIS.IS.A.FUNCTION EXTERNAL
    INTEGER THIS.IS.A.FUNCTION
FEND
CALL THIS.IS.A.FUNCTION
```

In the example above no code is generated for the EXTERNAL function THIS.IS.A.FUNCTION. The compiler generates the required information so that the linkage editor will link the call to function THIS.IS.A.FUNCTION with its definition in another module.

If two modules are linked together only those functions which are public in one module and external in another are linked. Each module may use the same name for functions which are not PUBLIC or EXTERNAL without confusion.

Parameters may be passed to external functions in the same manner as they would be passed to a procedure defined in the same module in which they are accessed. No type checking is performed when parameters are passed to an external procedure. It is the programmers responsibility to ensure that corresponding parameters agree in type.

```
DEF ADD(A,B) PUBLIC
     STRING A,B,ADD

     ADD = A + B
FEND
```

ADD is a public procedure. It may be accessed from another module by using the following external function definition:

```
DEF ADD(STR1$,STR2$) EXTERNAL
     STRING ADD
FEND
```

Note that the parameter names do not have to be the same. However the function names must be the same and the types of the parameters must agree. An equivalent definition would be:

```
DEF ADD(S1,S2) EXTERNAL
     STRING ADD,S1,S2
FEND
```

Some linkage editors may restrict the length of external names. See appendix E for current restrictions.

11.2. Linkage With Assembly Language Routines

An external function does not have to be generated by another CB8Ø program. It could be an assembly language program. The only requirement is that the assembly language program must observe the CB8Ø parameter passing conventions. All parameters are passed on the stack. Integers and real numbers are placed on the stack directly. In the case of strings, a pointer to the string is placed on the stack.

Integers and strings each occupy two bytes on the stack. The values are stored as sixteen bit addresses with the low-order byte first. Real numbers are stored as eight byte quantities. The top byte on the stack is the exponent. The eighth byte is the most significant byte of the mantissa. Refer to chapter three for more information on the format of data items.

If the address corresponding to a string parameter is zero, the string is a null string. Otherwise the address points to the string. The first two bytes of the string represent the length of the string, with the high-order byte first.

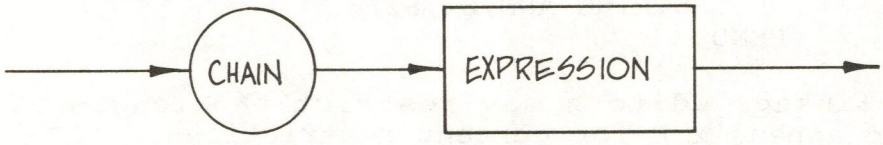
If the high-order bit of the string length is a one, the string is a temporary string. The space for temporary strings must be released prior to returning from an assembly language function. The method of releasing strings is machine dependent. The LK8Ø reference manual will provide information on releasing strings.

The SADD function, explained in chapter six, may be used to pass the location of a string without having to worry about whether a string is temporary.

11.3. Chaining to Another Program

The CHAIN statement loads a new program and then executes the program. Two types of programs may be loaded: an overlay program created by the linkage editor, or a directly executable core image file.

The information concerning the CHAIN statement is general, and specific examples apply to the CP/M operating system. For more detailed information on linking modules and programs, refer to the linkage editor documentation.



The expression, which must evaluate to a string, is the name of the program that is to be loaded. If no type extension is specified, a type of OVL is assumed. An execution error occurs if the file cannot be opened.

```
CHAIN "RPTWRT"
```

The statement above will load the file "RPTWRT.OVL" and then execute the new program. All "OVL" files loaded by a CHAIN statement must have been linked with the last "COM" file loaded.

CHAIN "AR.COM"

This statement loads and executes the file "AR.COM". When a program is loaded, the variables in the data area are set to zero if they are numeric and to null strings if they are string variables. Any variables in the COMMON area remain as they were before the chain statement was executed.

If the program being chained to has a type extension of "COM", and it has a different name then the last "COM" file loaded, the COMMON variables are also reset to zero or null strings. This allows a CHAIN statement to load and execute a completely new application.

A CHAIN statement may load a "COM" file created by languages other than CB8Ø. The "COM" files loaded need not be created by LK8Ø. However, all "OVL" files loaded must have been created by LK8Ø. In addition if a "COM" file chains to an "OVL" file both the "COM" and "OVL" files must have been created by LK8Ø.

The CB8Ø runtime support system zeros the data area prior to executing a program. This means that assembly language modules linked with CB8Ø modules cannot have initialized data in data segments.

12. COMPILER OPERATION

This chapter describes how to use the CB8Ø compiler to compile source programs. In addition it explains the workspace requirements of the compiler and the toggles which modify compiler operation.

12.1. Compiling a Program

The CB8Ø compiler is executed using the following command:

```
CB8Ø TEST
```

TEST is the name of the source program which has a default type extension of BAS. This command will compile TEST, generate a relocatable object file and list the program on the console. The listing will provide a line number, the relative address of the code generated by the line, and the actual source line.

The CB8Ø compiler includes three overlays:

```
CB8Ø.OV1
CB8Ø.OV2
CB8Ø.OV3
```

All of the overlays must be on the same logical device as the executable module:

```
CB8Ø.COM
```

The source file may be on any logical disk device. For example:

```
CB8Ø D:TEST
```

will compile the program TEST.BAS from drive D:.

The default extension of "BAS" may be overridden by specifying a complete file name.

```
CB8Ø TEST.PR1
```

The above command will compile the program TEST.PR1.

The compiler will create work files with an extension of "TMP" on the same device as the source file unless a drive is specified by a compiler toggle described below. The following

temporary files are used by CB8Ø:

PA.TMP

QCODE.TMP

DATA.TMP

If any files with these names exist on the workfile disk they will be deleted by CB8Ø. In addition CB8Ø will create a file with the same name as the source file and extension "REL" on the same device as the source file. If the source program contains errors a relocatable "REL" file is not created.

The size of the "TMP" files will vary from program to program but the amount of temporary space required is approximately the same amount as the source files being compiled. The REL file will also be about the same size as the source file.

On systems with limited disk space it may be necessary to break the program into modules and compile each module separately.

12.2. Compiler Toggles

The command line which invokes the compiler may pass information to the compiler by using compiler toggles. The toggles are alphabetic characters enclosed in square brackets.

```
CB8Ø TEST [B]
```

The command above will compile TEST.BAS with the "B" toggle in effect. The toggles may be either lower or upper case letters. The source file name is automatically terminated when a left square bracket is encountered. The following commands have an identical effect as the one above:

```
CB8Ø TEST[B]

CB8Ø TEST[b]
```

In all cases the source file name is TEST.BAS.

If the source file cannot be located, an error message is printed and CB8Ø returns control to the operating system. The same message is printed if a %INCLUDE directive (chapter two) cannot find a source file.

Other file system or memory space errors result in a message being printed and compilation terminated. Appendix A lists these messages.

The following toggles are supported:

B	Suppress listing of the source file
C	Change the default Include file disk
D	Generate error for undeclared variables
I	Interlist the generated code with the source file
L	Set the page length
N	Generate code for line numbers
O	Suppress the generation of the object (REL) file
P	List the source file on the printer
S	Include symbol name information in the REL file
T	List the symbol table following the source file listing
W	Set the page width
X	Specify a disk for the work files

The B toggle suppresses all listing. Only the statistical data concerning the size of code and data areas is listed on the console. If errors are detected, the error and the source line containing the error are listed.

The B toggle overrides other toggles that would result in compiler output. The B toggle has the effect of starting the program with a %NOLIST compiler directive. The %LIST directive will override the B toggle.

The C toggle changes the default logical drive for INCLUDE files. If a %INCLUDE directive specifies a file name with no disk specified, the file would be normally included from the same drive as the source file.

The C toggle can override this assignment. For example the command:

```
CB8Ø TEST [c(d)]
```

will get INCLUDE files from drive D:. The required drive must be enclosed in parentheses. If the %INCLUDE directive specifies a drive, then the D toggle has no effect.

This toggle allows program development to be independent of the particular configuration of the hardware being used.

The D toggle will generate an error if a variable name does not appear in an INTEGER, REAL, or STRING declaration. This is used to locate misspelled identifiers and to improve documentation of a program.

The I toggle interlists the code that is generated with the source line that resulted in the code being generated. The generated code uses standard 8Ø8Ø nmemonics.

The L toggle can be used to change the page length. The new length must follow the L and be enclosed in parentheses. The length may be any unsigned integer constant.

```
CB8Ø TEST [L(4Ø)]
```

Initially the page length is set to 66.

The N toggle will cause code to be generated for each physical line in the source program. This allows the ERRL function to return the line number that an error occurred in.

The O toggle suppresses the generation of the relocatable object (REL) file. This will reduce, somewhat, the time to compile a program. The REL file is not created if errors are detected by the compiler.

The P toggle causes the listing to be directed to the printer. Each page has a heading with the page number and the source file name.

The S toggle causes information on program symbols to be included in the relocatable object (REL) file. The symbols can be used by the link editor to create a "SYM" file for debugging.

The T toggle results in a listing of the symbol table following the source file listing.

The W toggle can be used to change the width of output to the printer. The width is initially set to 8Ø columns. The new width must follow the W and be enclosed in parentheses. The width may be any unsigned integer constant.

```
CB8Ø TEST [W(72)]
```

The X toggle selects a drive for workfiles. If there is no X toggle specified the workfiles are placed on the same drive as the source file. The required drive must be enclosed in parentheses. It may be either an upper or lower case letter.

```
CB8Ø TEST [X(B)]
```

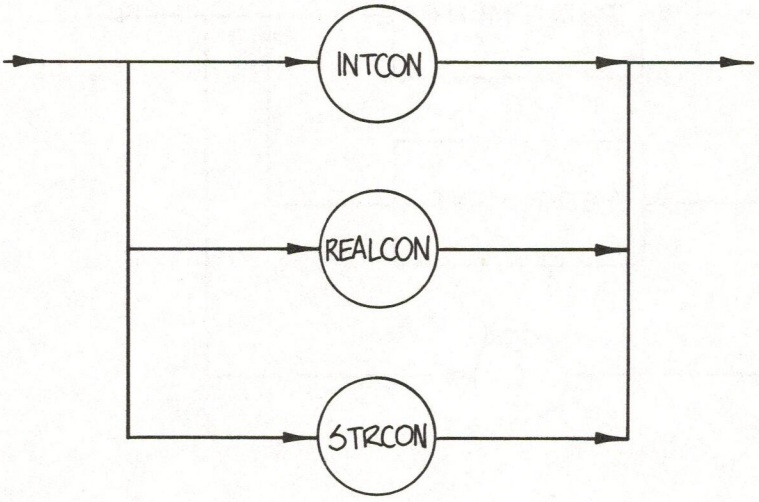
A. CB8Ø RESERVED WORDS

ABS	AND	AS	ASC	ATN
BUFF	CALL	CHAIN	CHR$	CLOSE
COMMAND$	COMMON	CONCHAR%	CONSOLE	CONSTAT%
COS	CREATE	DATA	DEF	DELETE
DIM	ELSE	END	ERR	ERRL
EXP	EXTERNAL	FEND	FLOAT	FOR
FRE	GO	GOSUB	GOTO	IF
INITIALIZE	INKEY	INP	INPUT	INT
INT%	LEFT$	LEN	LET	LINE
LOCK	LOG	LPRINTER	MATCH	MID$
MOD	NEXT	NOT	ON	OPEN
OR	OUT	PEEK	POKE	POS
PRINT	PUBLIC	RANDOMIZE	READ	RECL
RECS	REM	REMARK	RENAME	RESTORE
RETURN	RIGHT$	RND	SADD	SGN
SIN	SIZE	SQR	STEP	STOP
STR$	TAB	TAN	THEN	TO
UCASE$	UNLOCK	USING	VAL	VARPTR
WEND	WHILE	WIDTH	XOR	
%EJECT	%INCLUDE	%LIST	%NOLIST	%PAGE

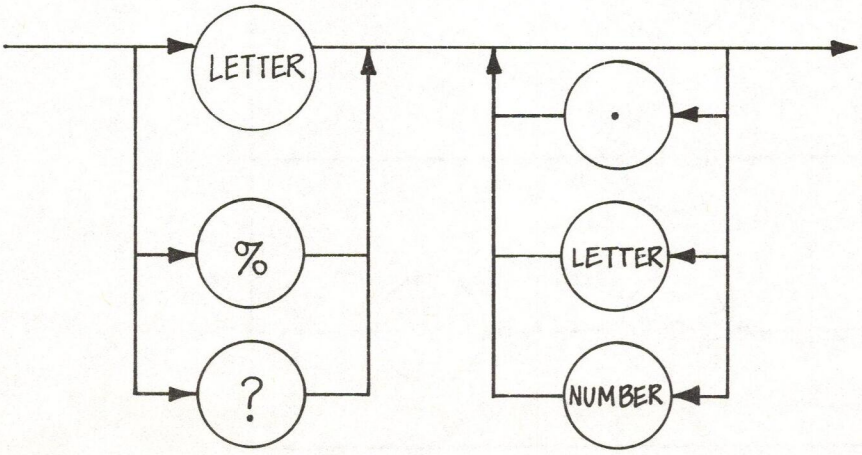
B. COLLECTED SYNTAX DIAGRAMS

This appendix contains the syntax diagrams describing the complete syntax of CB8Ø.

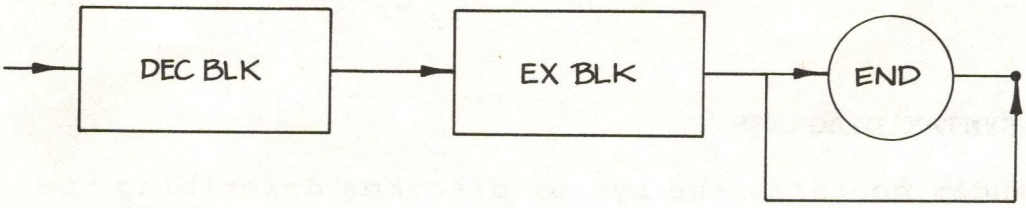
CONSTANT



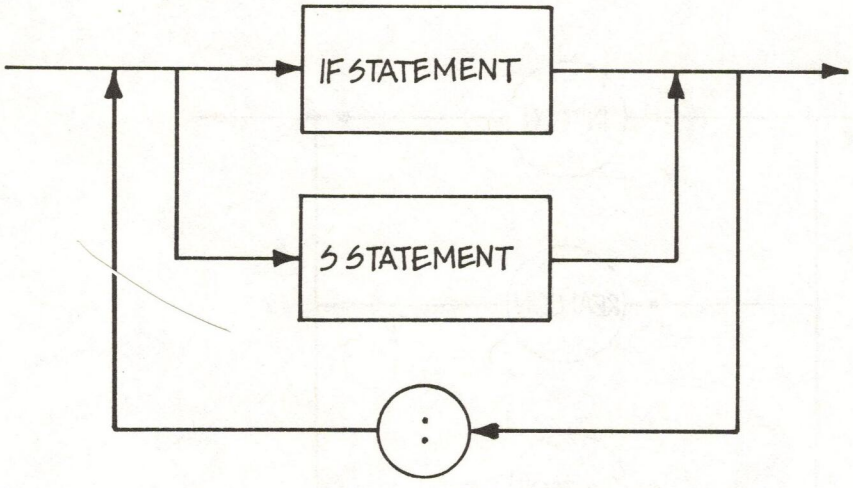
ID



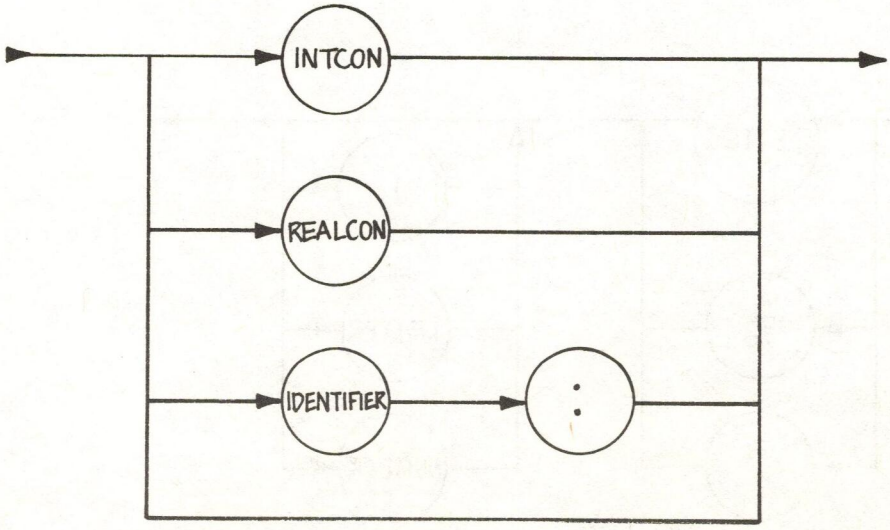
PROGRAM



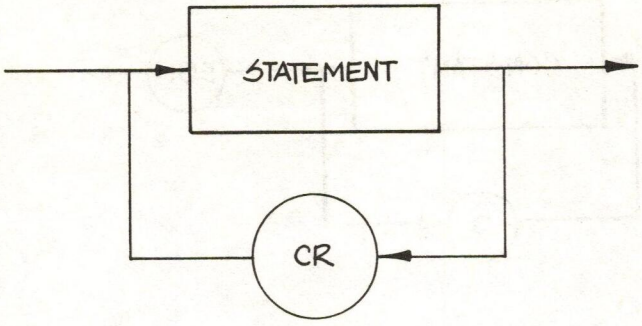
STATEMENT



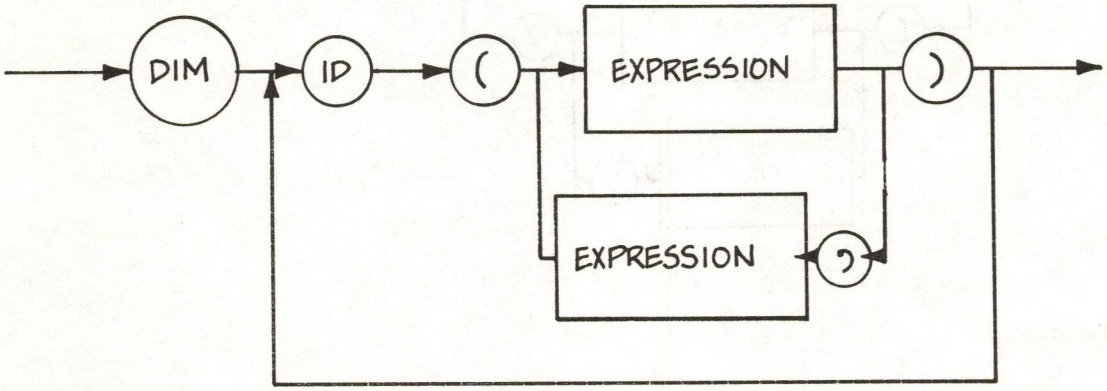
STATEMENT LABEL



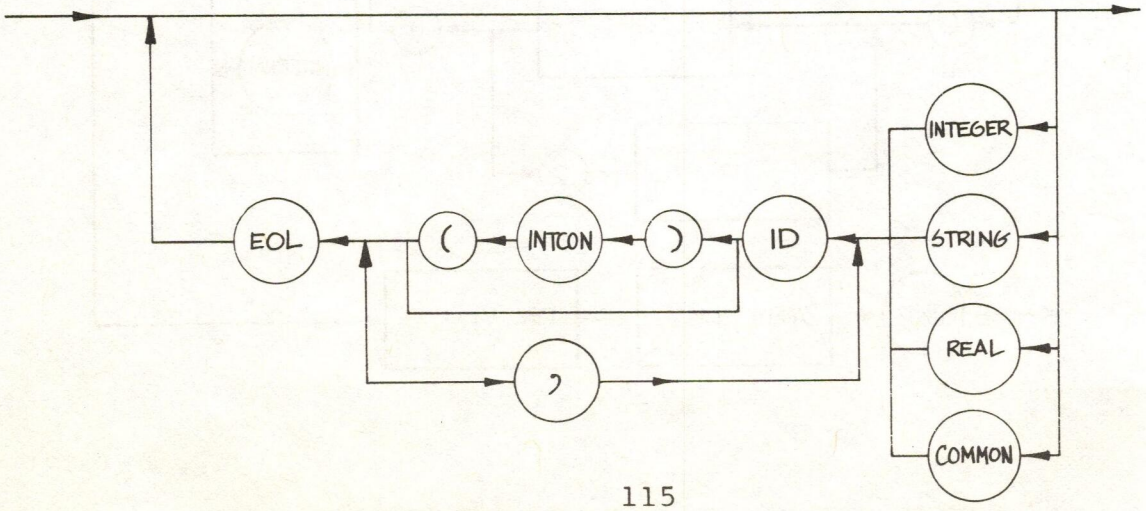
STMT BLK



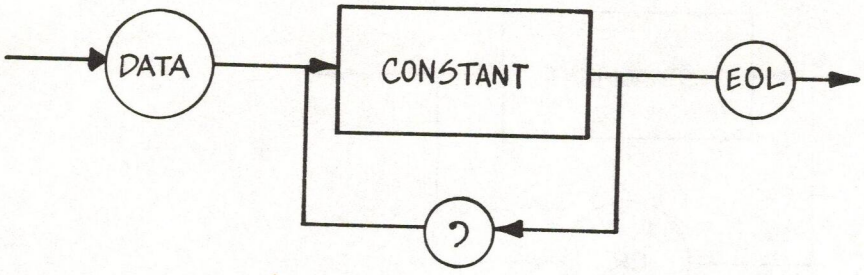
DIM



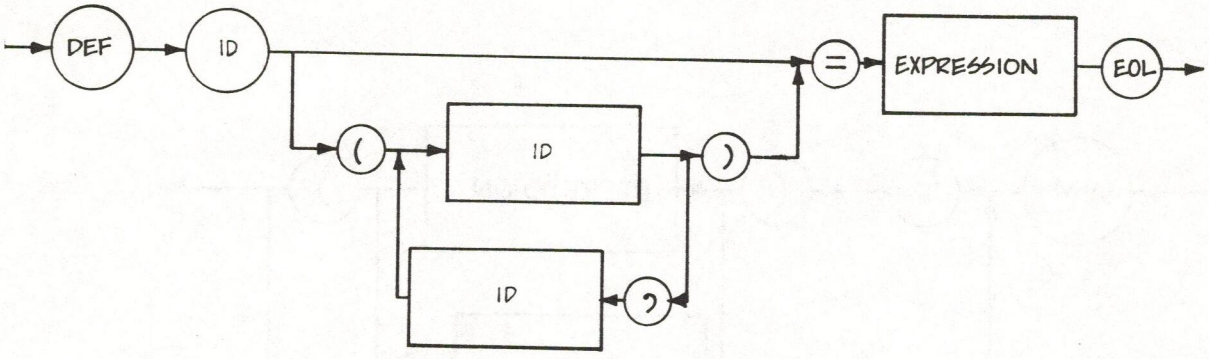
DECBLK



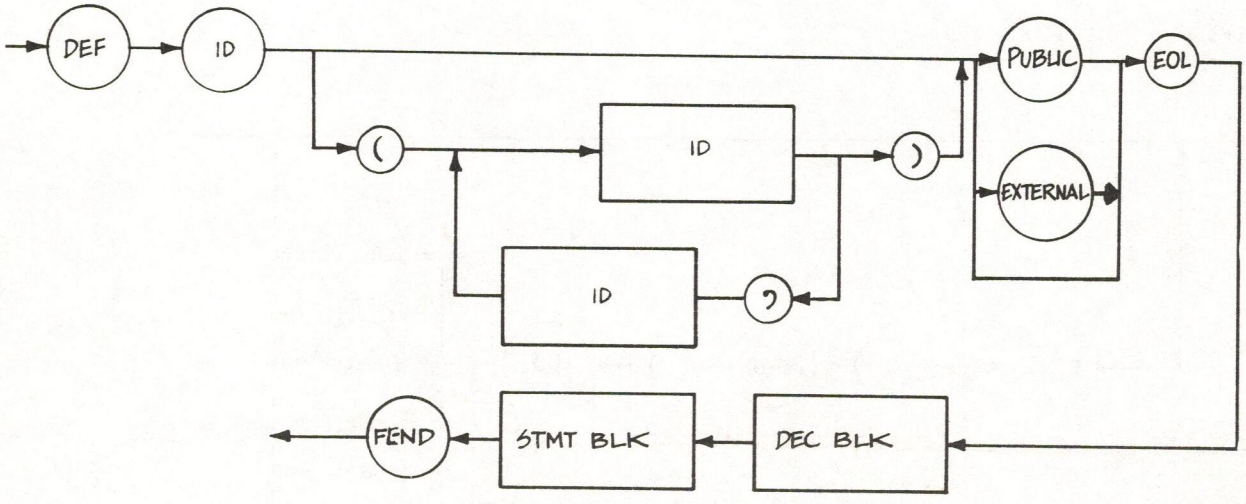
DATA



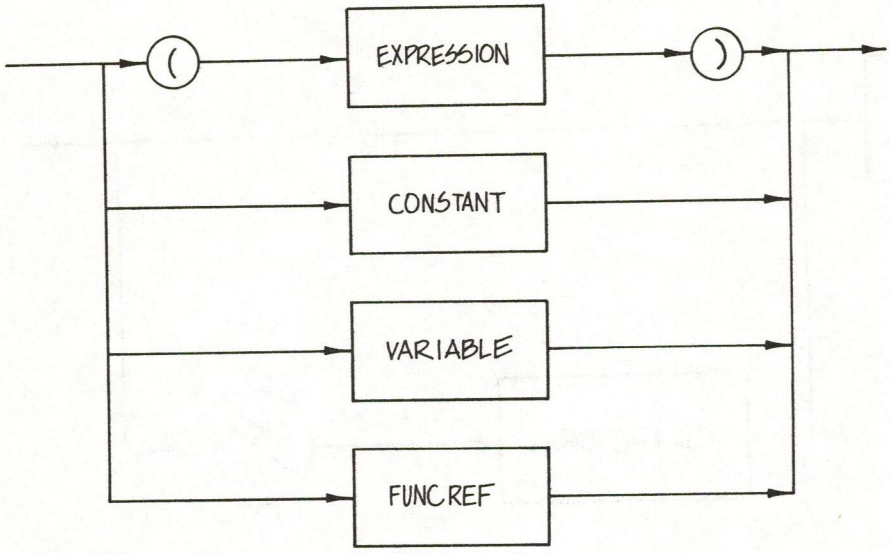
SINGLE LINE FUNCTION



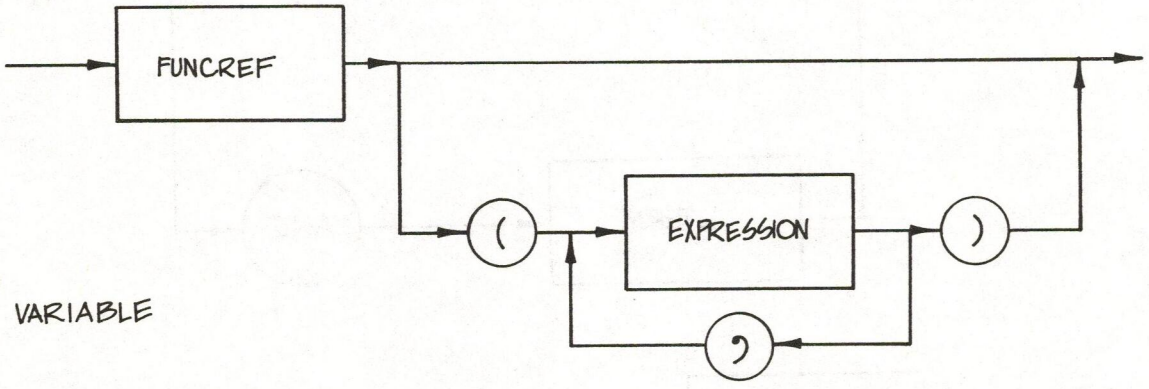
MULTIPLE LINE FUNCTION



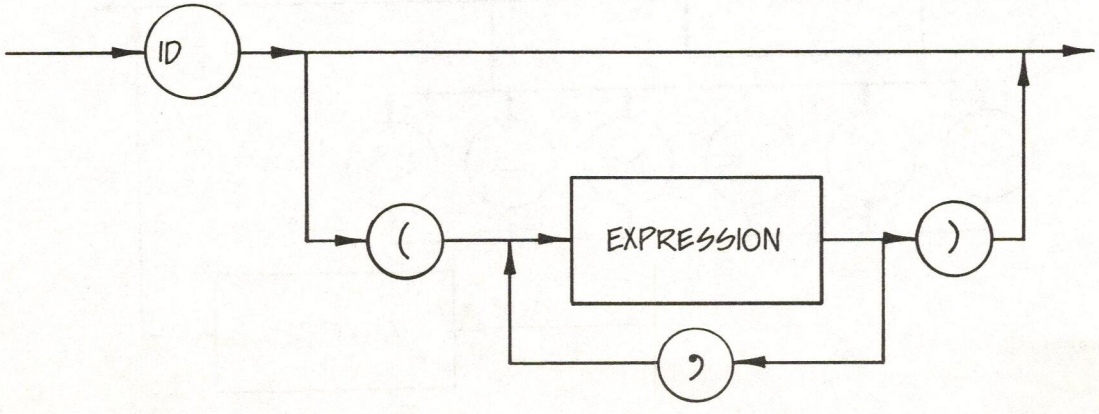
OPERAND



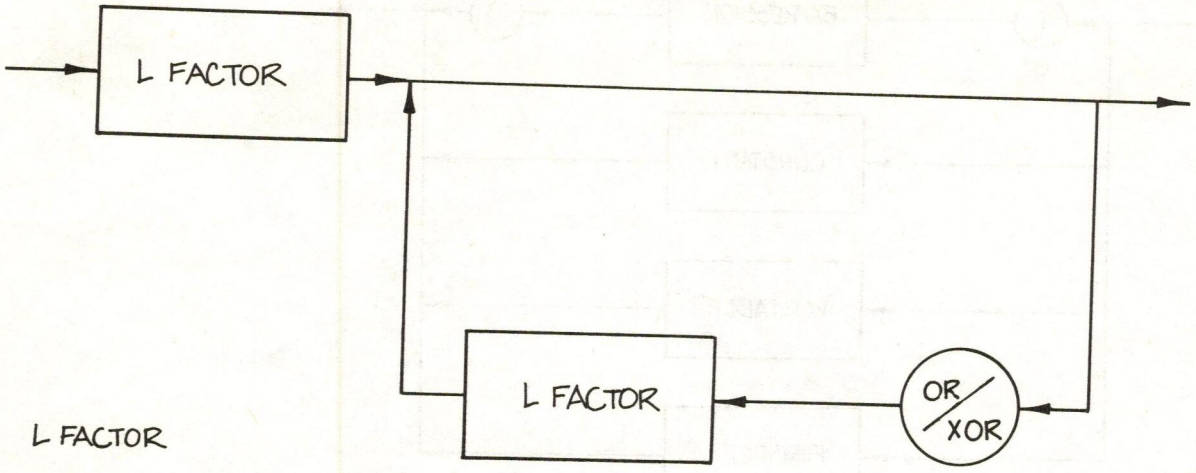
FUNCREF



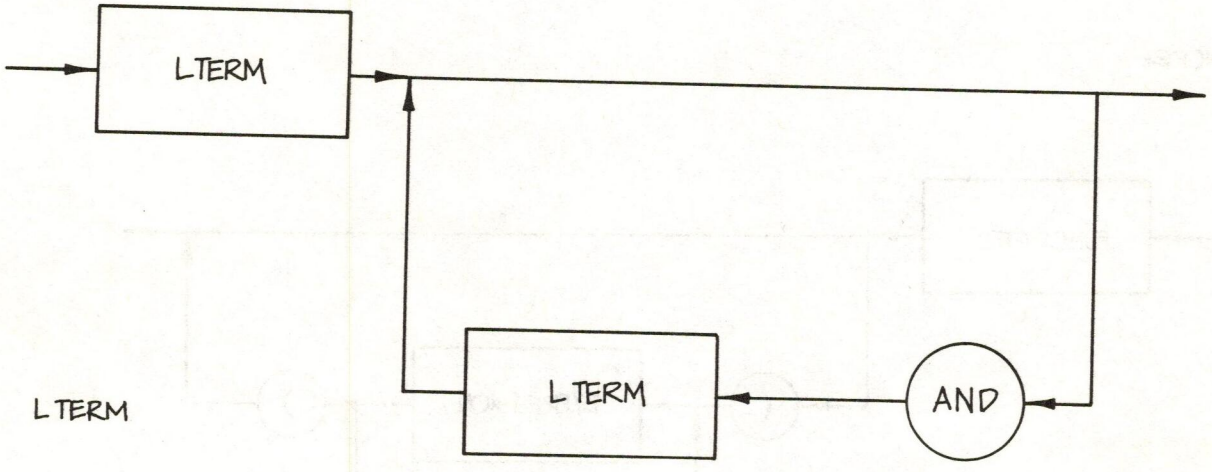
VARIABLE



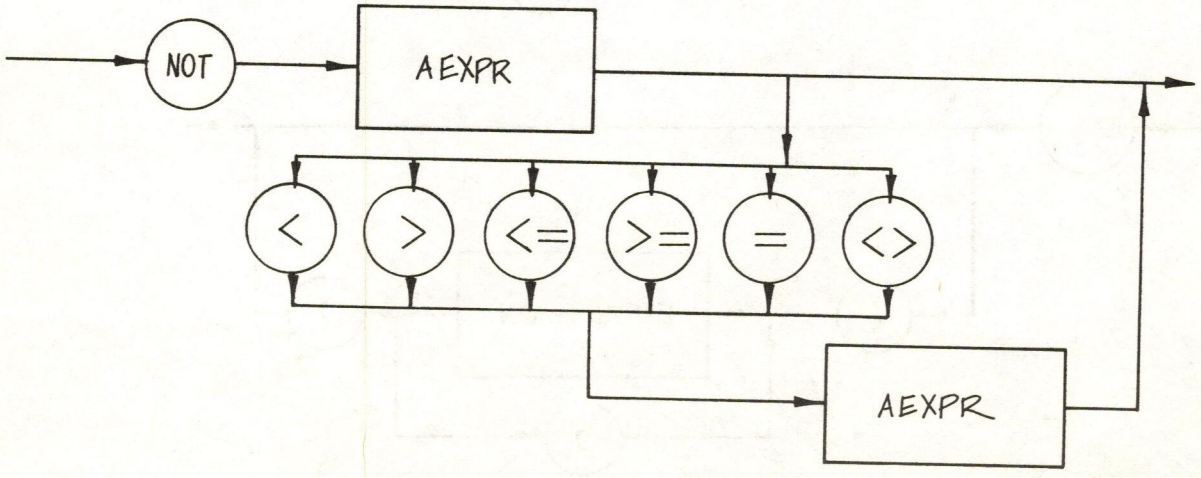
EXPRESSION



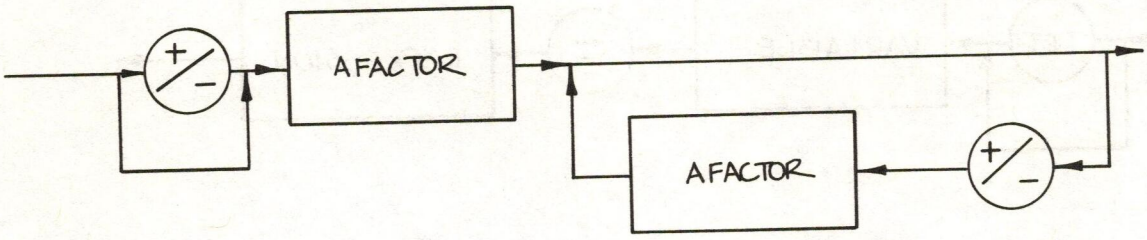
L FACTOR



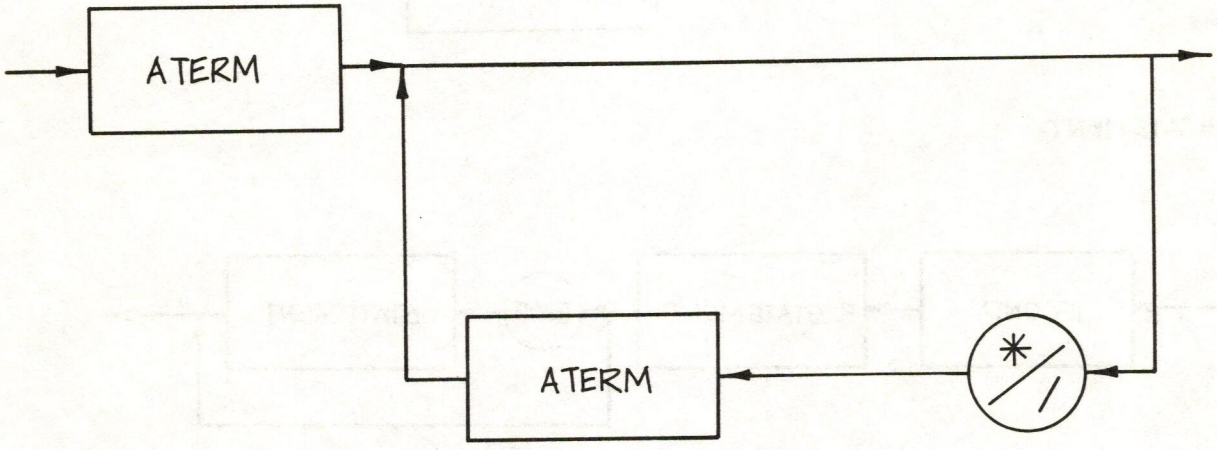
LTERM



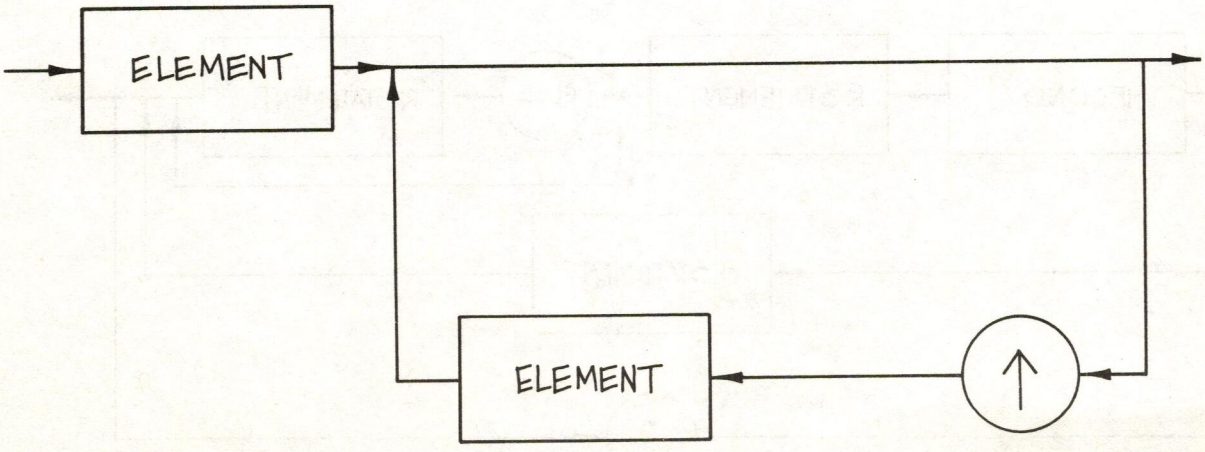
AEXPR



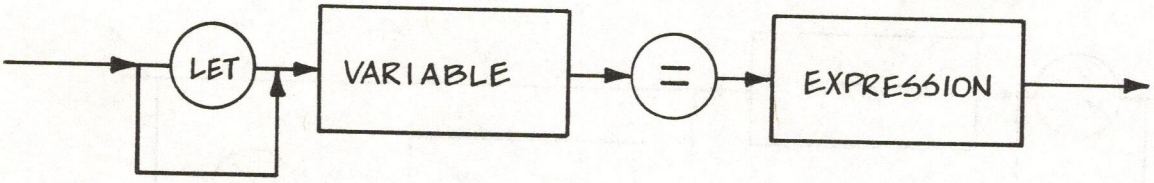
AFACTOR



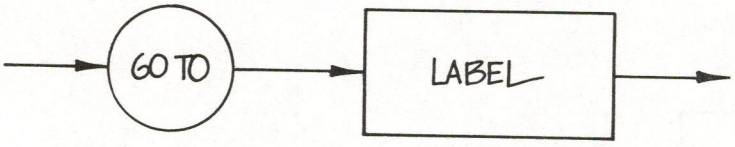
ATERM



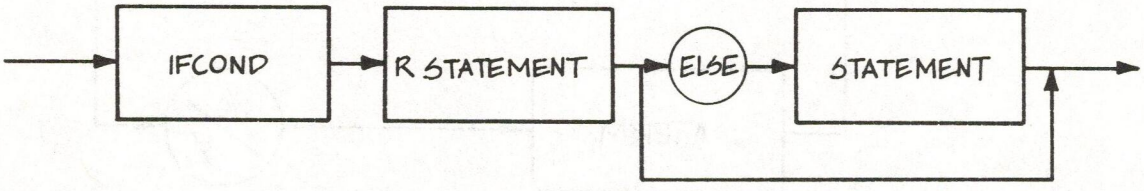
ASSIGNMENT STATEMENT



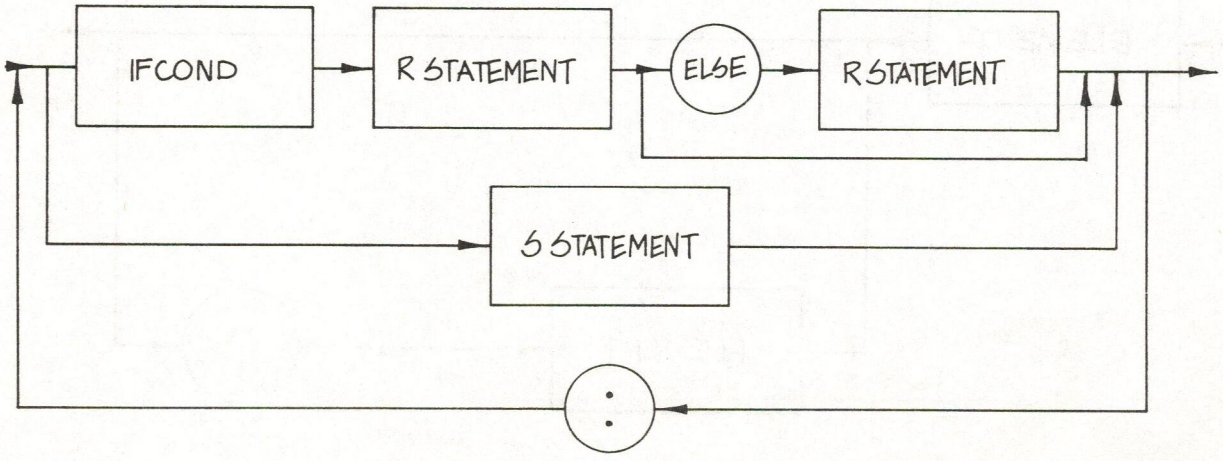
GO TO



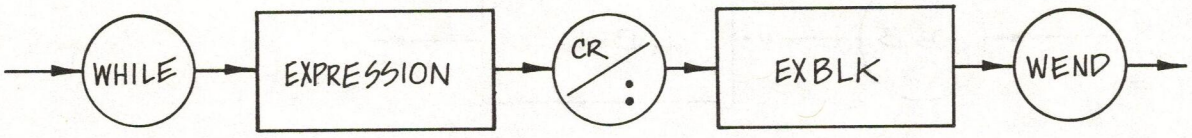
IF STATEMENT



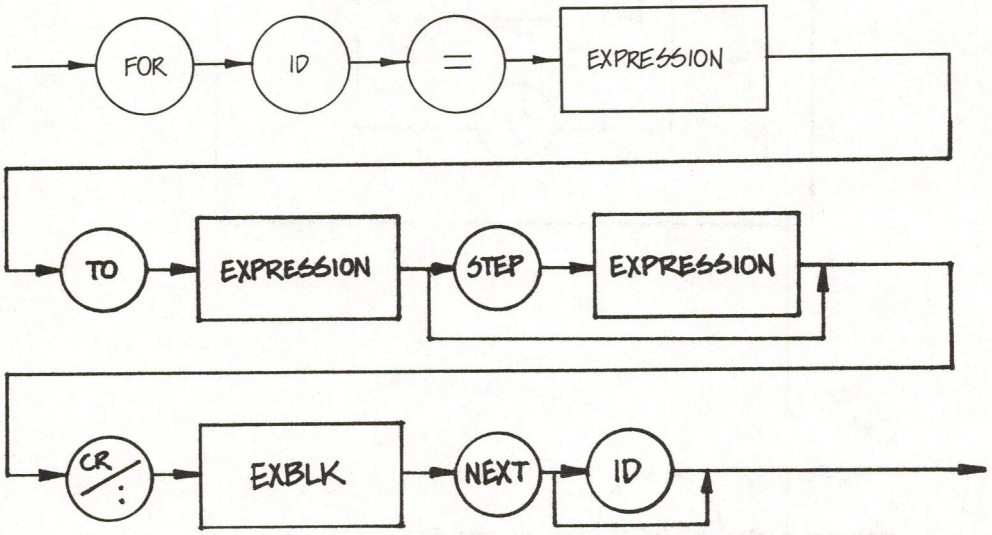
R STATEMENT



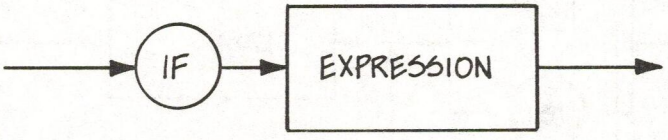
WHILE LOOP



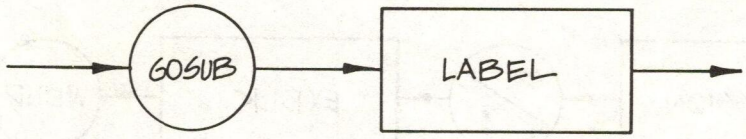
FOR LOOP



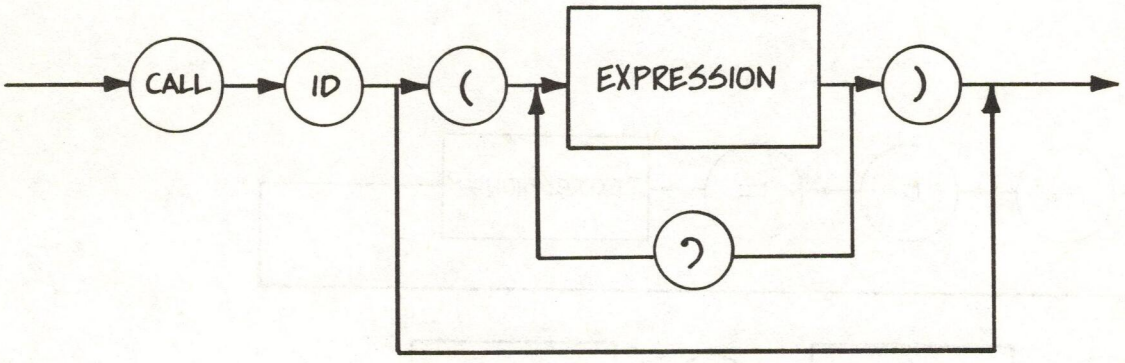
IFCOND



GO SUB

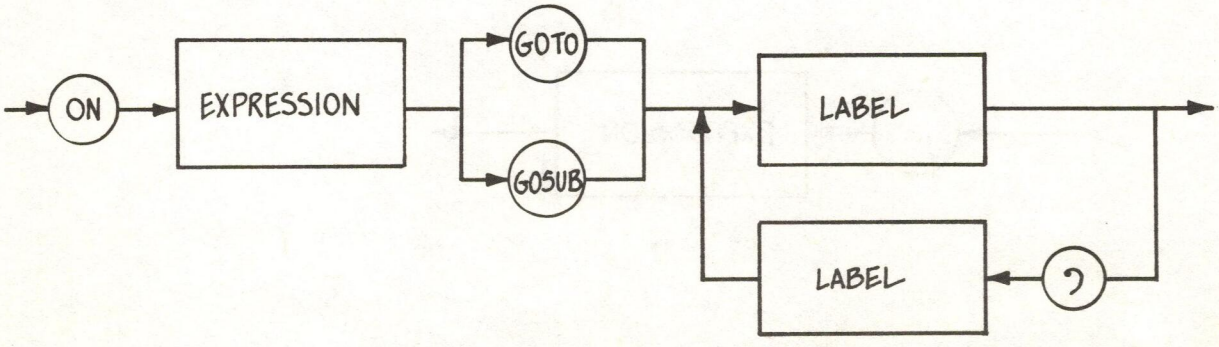


CALL

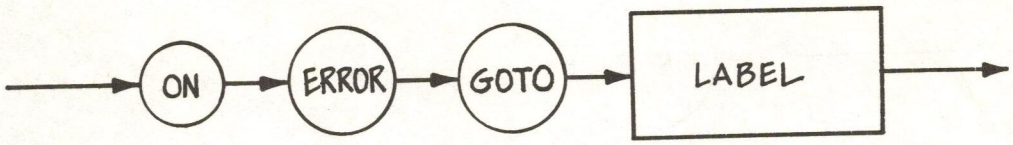


RETURN

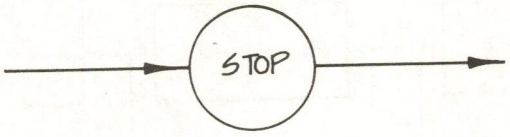
ON



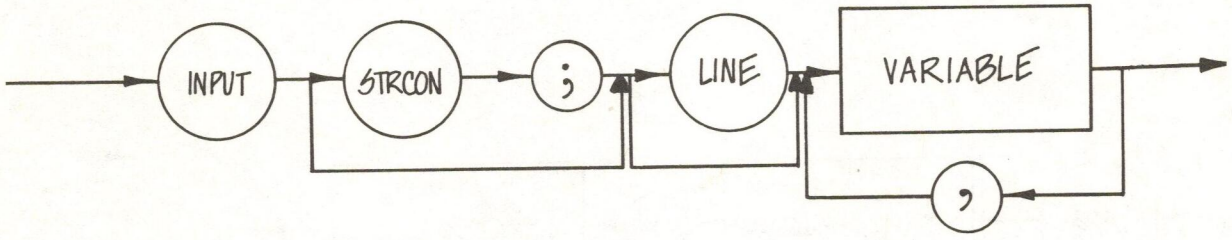
ON ERROR



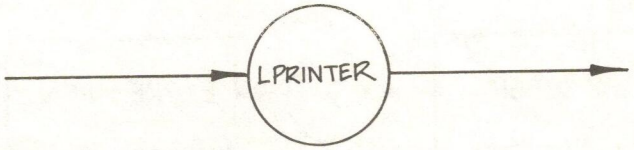
STOP



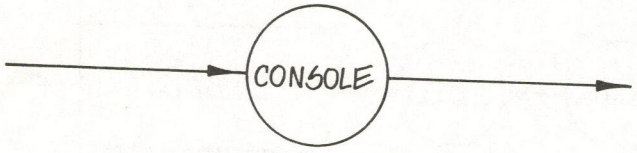
INPUT



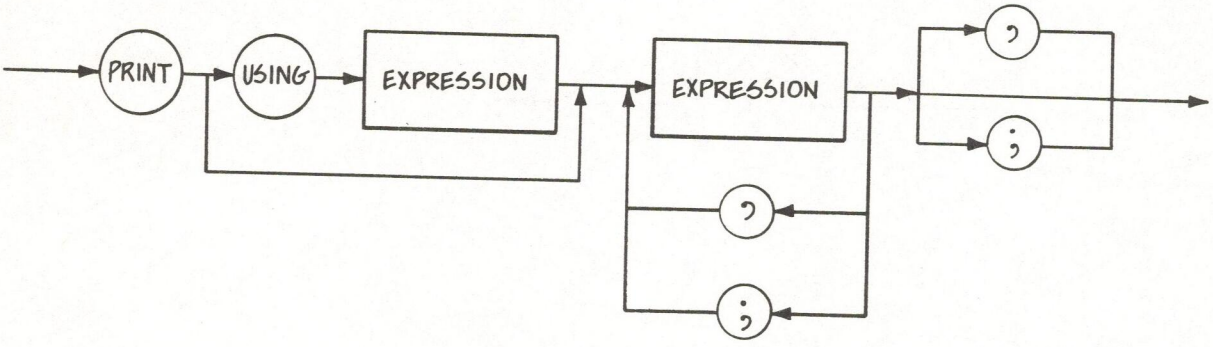
LPRINTER



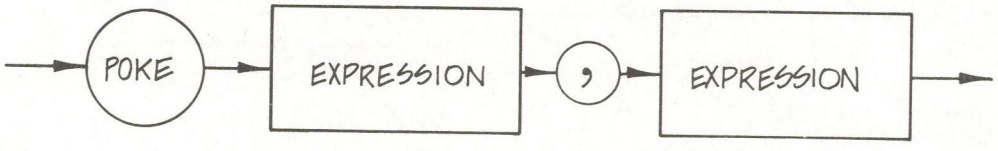
CONSOLE



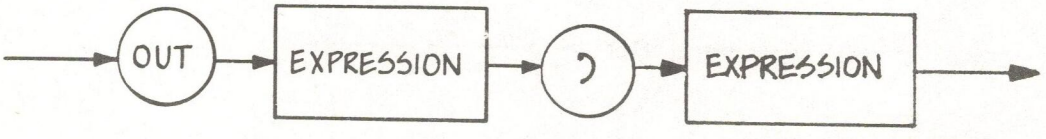
PRINT



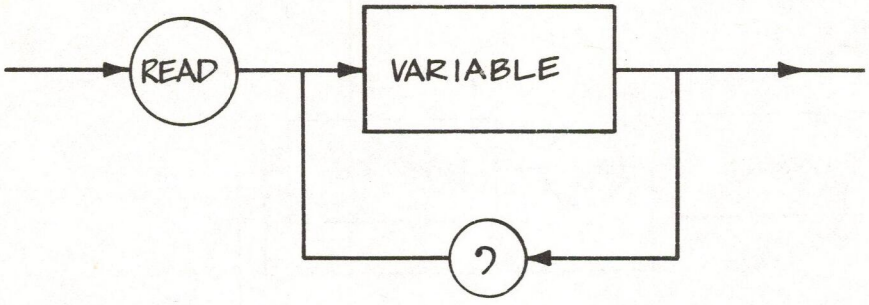
POKE



OUT

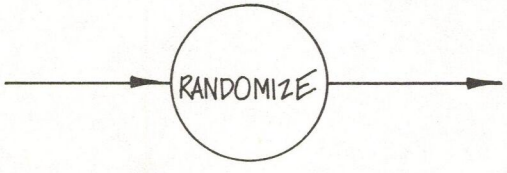


READ

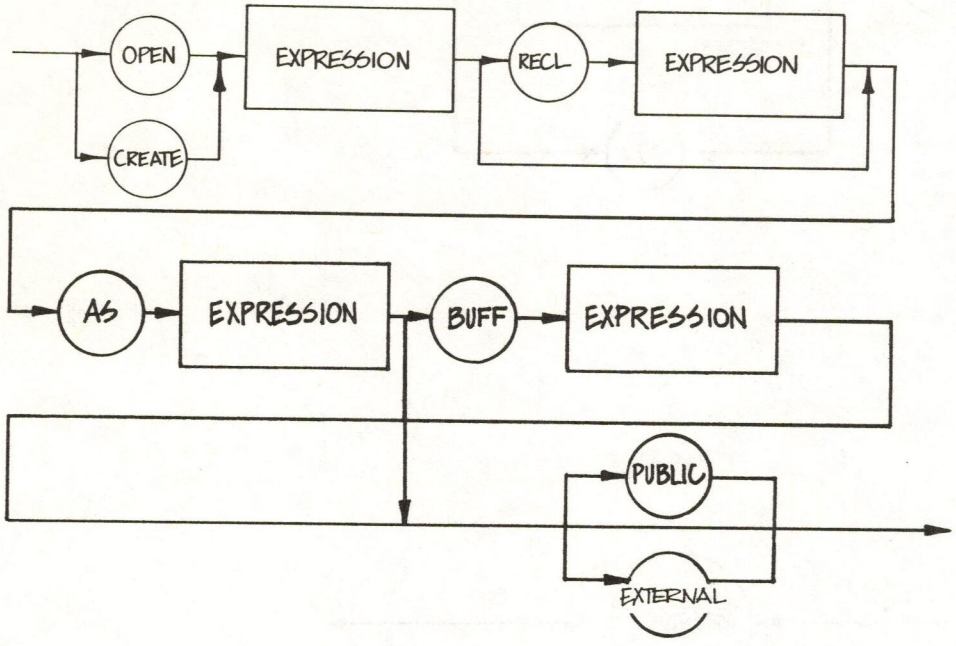


RESTORE

RANDOMIZE

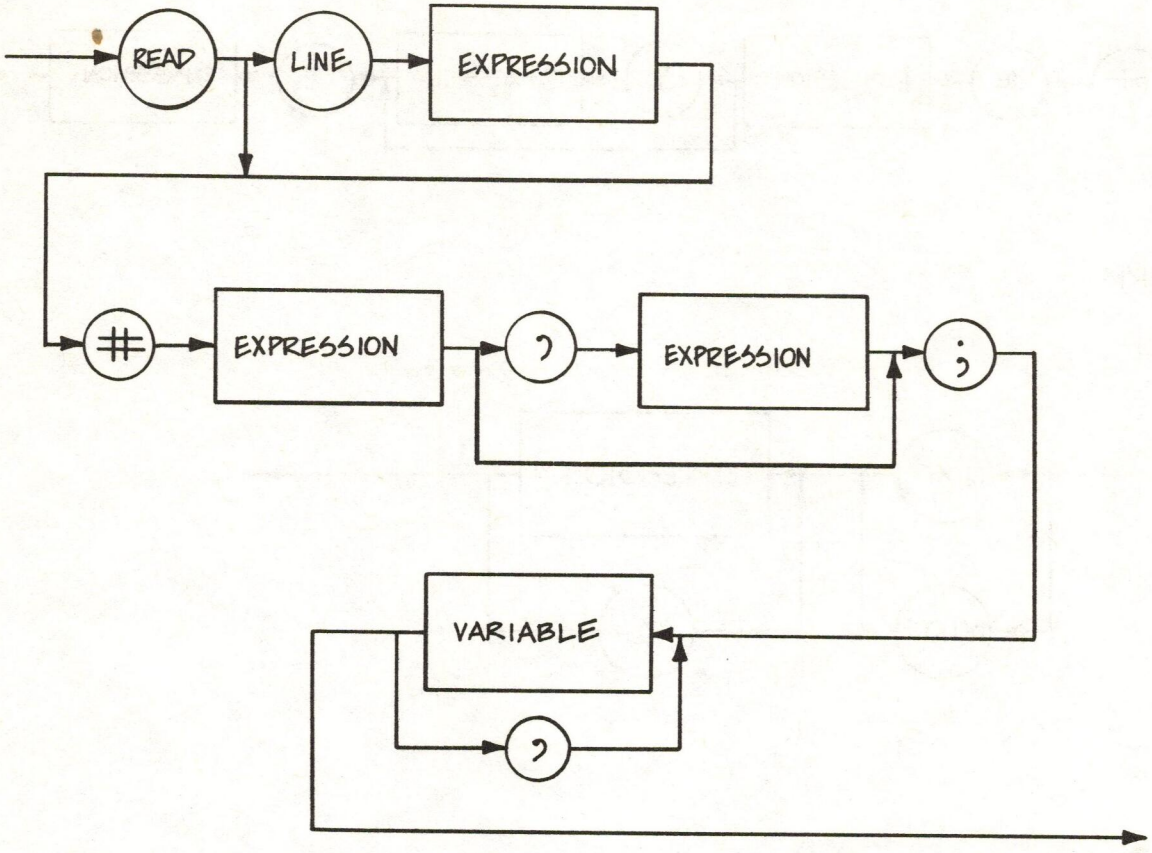


FACCESS

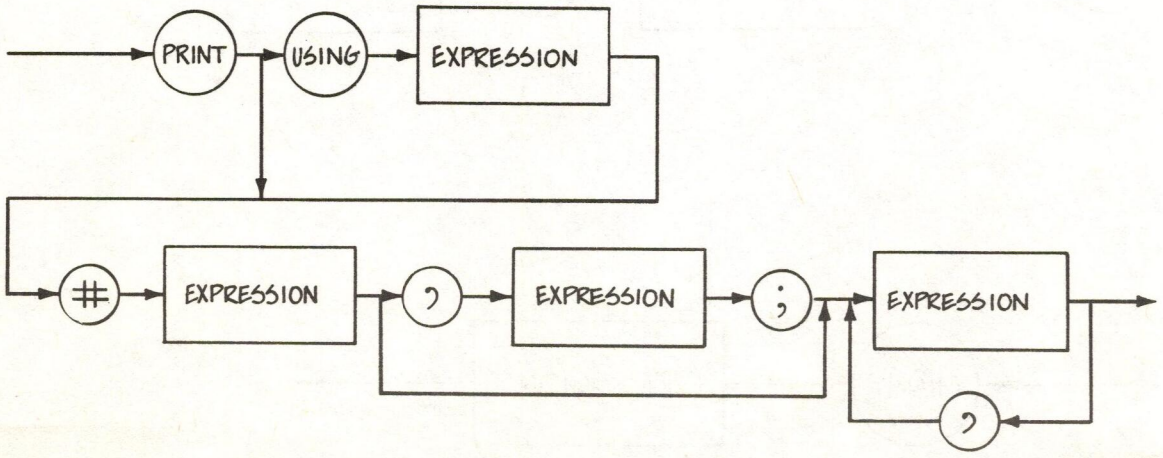


INIT

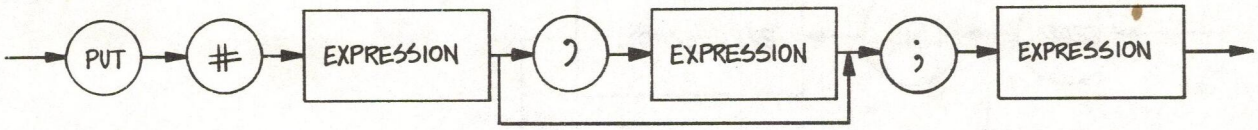
READ FILE



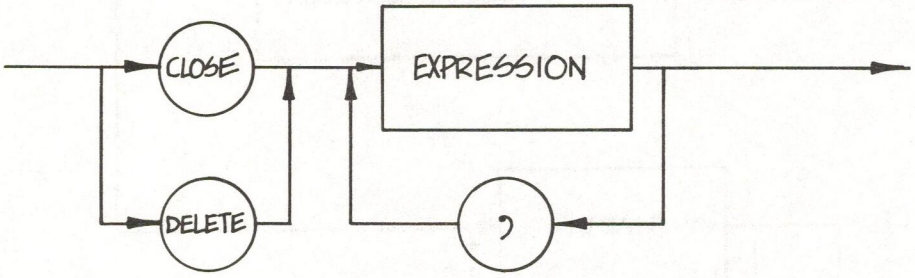
PRINT FILE



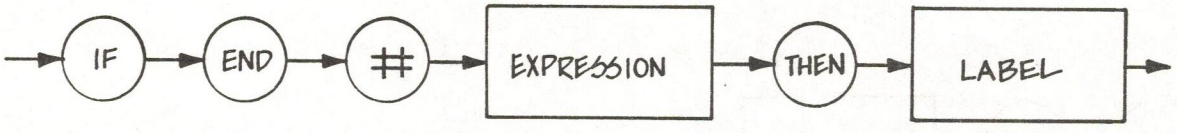
PUT



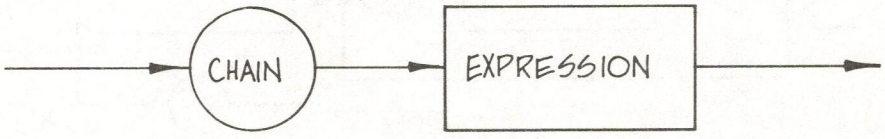
F TERM



IF END



CHAIN



C. COMPILER ERROR MESSAGES

The following messages are printed by the compiler when a file system error or memory space error occurs. In each case control is returned to the operating system.

COULD NOT OPEN FILE: <file name>

The file name following the message could not be located in the file system directory.

INCLUDES NESTED TO DEEP: <file name>

The file name following the message occurred in an %INCLUDE directive that would have exceeded the allowed nesting of %INCLUDE directives.

SYMBOL TABLE OVERFLOW

The available memory for symbol table space has been exceeded. Break the program into modules or use shorter symbol names.

INVALID FILE NAME: <file name>

The file name is not valid for the operating system being used.

DISK READ ERROR

The operating system reported a disk read error.

CREATE ERROR: <file name>

The file could not be created. Normally this means there was no directory space on the disk.

DISK FULL

The operating system reported that no additional space is available to write temporary or output files.

INVALID COMMAND LINE

The CB8Ø command line was improper.

CLOSE OR DELETE ERROR

The operating system reported it could not close a file. This could occur if diskettes were switched during compilation.

The following error messages may be generated during compilation of a program. Compilation continues after the error has been recorded.

1 An invalid character was detected in the source program. The character was ignored.

2 Invalid string constant. The string is too long or contains a carriage return.

3 Invalid numeric constant. An integer constant of zero is assumed.

4 Undefined compiler directive. This source line is ignored.

5 The %INCLUDE directive is missing a file name. This source line is ignored.

6 Not used.

7 Not used.

8 A variable was used without being defined and the U toggle was used during compilation.

9 The DEF statement is not terminated by a carriage return. A carriage return was inserted.

1Ø A right parenthesis is missing from the parameter list. A right parenthesis was inserted.

11 A comma was expected in the parameter list. A comma was inserted.

12 An identifier was expected in the parameter list.

13 The same name is used twice in a parameter list.

14 A DEF statement occurred within a multiple line function. Multiple line functions may not be nested. The statement was ignored.

15 A variable was expected.

16 The function name was missing following the keyword DEF. The DEF statement was ignored.

17 A function name has been used previously. The DEF statement is ignored.

18 A FEND statement was expected. A FEND was inserted.

19 There are too many parameters in a multiple line function.

2Ø Inconsistent identifier usage. An identifier cannot be used as both a label and a variable.

21 Additional data exists in the source file following an END statement. This is the logical end of the program.

22 Data statements must begin on a new line. The remainderof this statement was treated as a remark.

23 A reserved word appears in a declaration list. The reserved word was ignored.

24 A function name appears in a declaration within a multiple line function other than the multiple line function that defines this function name.

25 A function call was encountered with the incorrect number of parameters.

26 A left parenthesis was expected. A left parenthesis was inserted.

27 Invalid mixed mode. The type of the expression is not permitted.

28 Unary operator cannot be used with this operand.

29 Function call with improper type of parameter.

3Ø Invalid symbol following a variable, constant, or function reference.

31 This symbol cannot occur at this location in an expression. The symbol is ignored.

32 Operator is missing. Multiplication operator inserted.

33 Invalid symbol encountered in an expression. The symbol is ignored.

34 A right parenthesis was expected. A right parenthesis inserted.

35 A subscripted variable is used with the incorrect number of subscripts.

36 An identifier is used as a simple variable with previous usage as a subscripted variable.

37 An identifier is used as a subscripted variable with previous usage as an unsubscripted variable.

38 A string expression is used as a subscript in an array reference.

39 A constant was expected.

4Ø Invalid symbol found in declaration list. The symbol is skipped.

41 A carriage return was expected in a declaration statement. A carriage return was inserted.

42 Comma is missing in declaration list. A comma inserted.

43 A common declaration may not occur in a multiple line function. The statement is ignored.

44 An identifier appears in a declaration twice in the main program or within the same multiple line function.

45 The number of dimensions specified for an array exceeds the maximum number allowed. A value of one wasused. This may generate additional errors in the program.

46 Right parenthesis missing in the dimension specification within a declaration. A right parenthesis was inserted.

47 The same identifier is placed in COMMON twice.

48 An invalid subscripted variable reference was encountered in a declaration statement. An integer constant is required. A value of 1 was used.

49 An invalid symbol following a declaration or the symbol in the first statement in the program is invalid. The symbol is ignored.

5Ø An invalid symbol was encountered at the beginning of a statement or following a label.

51 An equal sign was expected in assignment. An equal sign was inserted.

52 A name used as a label was previously used at this level as either a label or variable.

53 Unexpected symbol following a simple statement. The symbol was ignored.

54 A statement was not terminated with a carriage return. Text was ignored until the next carriage return.

55 A function name was used in the left part of an assignment statement outside of a multiple line function. Only when the function is being compiled may its name appear on the left of an assignment statement.

56 A predefined function name was used as the left part of an assignment statement.

57 In an IF statement a THEN was expected. A THEN was inserted.

58 A WEND statement was expected. A WEND was inserted.

59 A carriage return or colon was expected at the end of a WHILE loop header.

6Ø In a FOR loop header the index is missing. Thecompiler skipped to end of this statement.

61 In a FOR loop header a TO was expected. A TO was inserted.

62 An equal sign was missing in a FOR loop headerassignment. An equal sign was inserted.

63 Expected carriage return or colon at end of FOR loop header.

64 A NEXT statement was expected. A NEXT was inserted.

65 Not used.

66 The variable which follows NEXT does not match the FOR loop index.

67 A NEXT statement was encountered without a corresponding FOR loop header.

68 A WEND statement was encountered without a corresponding WHILE loop header.

69 A FEND statement was encountered without a corresponding DEF statement. This error indicates that the end of the source program was detected while within a multiple line function.

7Ø The PRINT USING string is not of type string.

71 A delimiter is missing in a PRINT statement. A semicolon was inserted.

72 A semicolon was expected in an INPUT prompt. A semicolon was inserted.

73 A delimiter is missing in an INPUT statement. A comma was inserted.

74 A semicolon was expected following a file reference. A semicolon was inserted.

75 The prompt in an INPUT statement was not of type string.

76 In an INPUT LINE statement the variable following the keyword LINE was not a string variable.

77 In an INPUT statement a comma was expected between variables. A comma was inserted.

78 The keyword AS was missing in an OPEN or CREATE statement. AS was inserted.

79 The file name in an OPEN or CREATE statement was not a string expression.

8Ø A delimiter is missing in a READ statement. A comma was inserted.

81 In a GOTO, GOSUB or ON statement a label was expected. This token may be an identifier previously used as a variable.

82 The label in a GOTO statement is not defined. If the label is used in a function, it must be defined in that function.

83 A delimiter is missing in a file READ statement. A comma was inserted.

84 In a READ LINE statement the variable following the keyword LINE is not a string variable.

85 The label in an IF END statement is not defined.

86 A pound sign (#) was expected in an IF END statement. A pound sign was inserted.

87 A THEN was expected in an IF END statement. A THEN was inserted.

88 In a PRINT statement the semicolon is missing following a using string. A semicolon was inserted.

89 In an ON statement a GOTO or GOSUB was expected. A GOTO was assumed.

9Ø The index of a FOR loop header is of type string. The index must be an integer or real value.

91 The expression following the keyword TO in a FOR loop header is of type string. The expression must be an integer or real value.

92 The expression following the keyword STEP in a FOR loop header is of type string. The expression must be an integer or real value.

93 A variable in a DIM statement has been defined previously as other than a subscripted variable.

94 An identifier was expected as an array name in a DIM statement. The entire statement was ignored.

95 A left parenthesis was expected in a DIM statement. A left parenthesis was inserted.

96 A right parenthesis was expected in a DIM statement. A right parenthesis was inserted.

97 The maximum number of dimensions allowed with a subscripted variable was exceeded.

98 A comma was expected in a POKE statement. A comma was inserted.

99 The index of a FOR loop header was not a simple variable.

1ØØ In a call statement a multiple line function name was expected.

1Ø1 A file PRINT statement was terminated with a comma or semicolon.

1Ø2 A DIM statement is missing for this subscripted variable.

1Ø3 Expected a comma in the label list associated with an ON GOTO or ON GOSUB statement. A comma was inserted.

1Ø4 Expected a GOTO in an ON ERROR statement. A GOTO was inserted.

1Ø5 Expected a comma in a PUT statement. A comma was inserted.

1Ø6 The expression in an IF statement was of type string. An integer or real expression is required.

1Ø7 The expression in a WHILE loop header was of type string. An integer or real expression is required.

1Ø8 In an OPEN or CREATE statement the file name was missing.

1Ø9 In an OPEN or CREATE statement the expression following the reserved word AS was missing.

D. EXECUTION ERROR MESSAGES

The following warning message may be printed during execution of a CB8Ø program:

IMPROPER INPUT - REENTER

This message occurs when the fields entered from the console do not match the fields specified in the INPUT statement. This can occur when field types do not match or the number of fields entered is different from the number of fields specified. Following this message all values required by the input statement must be reentered.

Execution errors cause a two-letter code to be printed. The following list contains valid CB8Ø error codes. If an error occurs with a code consisting of an asterisk followed by a letter such as '*R', a CB8Ø library has failed. Please notify Compiler Systems of the circumstances under which the error occurred.

AC The argument in an ASC function was a null string.

BN The value folowing the BUFF option in an OPEN or CREATE statement is less than 1 or greater than 128.

CE The file being closed could not be found in the directory. This could occur if the file had been changed by the RENAME function.

CM The file specified in a CHAIN statement could not be found in the selected directory. If no type extension is present, a Itype of OVL is assumed.

CT The type extension of the file specified in a CHAIN statement was other than COM or OVL.

CU A close statement specified a file identification number that was not active.

DF An OPEN or CREATE statement used a file identification number that was already being used.

DU A DELETE statement specified a file identification number that was not active.

DW The operating system reported that there was no disk or directory space available for the file being written to and no IF END statement was in effect for the file identification number.

DZ A division by zero was attempted.

EF An attempt was made to read past the end of file and no IF END statement was in effect for the file identification number.

ER An attempt was made to write a record of length greater than the maximum record size specified in the OPEN or CREATE statement for this file.

FR An attempt was made to rename a file to a file name that already exists.

IF A file name used in an OPEN or CREATE statement or with the RENAME function was invalid for the operating system being used.

IR A record number of zero was specified in a READ or PRINT statement.

LN The argument in the LOG function was zero or negative.

ME The operating system reported an error during an attempt to create or extend a file. Normally this means the disk directory is full.

MP The third parameter in a MATCH function was zero or negative.

NE A negative value was specified for the operand to the left of the power operator.

NF A file identification was less than 1 or greater than the maximum number allowed. See appendix E.

NN An attempt was made to print a numeric expression with a PRINT USING statement but there was not a numeric field in the USING string.

NS An attempt was made to print a string expression with a PRINT USING statement but there was not a string field in the USING string.

OD A READ statement was executed but there are no DATA statements in the program, or all data items in all the DATA statements have already been read.

OE An attempt was made to OPEN a file that did not exist and for which no IF END statement is in effect.

OF An overflow occurred during a real arithmetic calculation.

OM The program ran out of dynamically allocated memory during execution.

RB Random access was attempted to a file activated with the BUFF option specifying more than one buffer.

RE An attempt was made to read past the end of a record in a fixed file.

RU A random read or print was attempted to a stream file.

SL A concatenation operation resulted in a string greater than the maximum allowed string length.

SQ A attempt was made to calculate the square root of a negative number.

SS The second parameter of a MID$ function was zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ was negative.

TL A tab statement contained a parameter less than 1.

UN A PRINT USING statement was executed with a null edit string or an escape char (\) was the last character in an edit string.

WR An attempt was made to write to a stream file after it had been read, but before it had been read to the end of file.

E. IMPLEMENTATION DEPENDENT VALUES

The following implementation dependent values apply to CB8Ø version 1.ØØ for use with CP/M version 2 and MP/M-8Ø versions 1 and 2:

Parameter	Value	Minimum
Initial page width for compiler output	8Ø	-
Initial page length for compiler output	66	-
Maximum number of errors maintained	95	-
Maximum nesting of include	6	4
Maximum number of formal parameters	15	15
Maximum number of subscripts in an array	15	15
Maximum unique identifier length	5Ø	31
Maximum number of characters in string constant	255	255
Maximum length of Global and External names	6	6
Maximum nesting of FOR loops	13	-
Maximum nesting of WHILE loops	39	-
Number of files that can be open at one time	2Ø	12
File buffer size in bytes	128	-

The minimum values are the minimum that will be used in any CB8Ø implementation.

The following extensions exist in CB8Ø version 1.ØØ to provide compatibility with CBASIC version 2.

The LPRINTER statement will accept a WIDTH option to be consistent with CBASIC. The width is ignored.

Integer and real data is initialized to Ø; strings are initialized to null strings.

The INPUT prompt string may be any expression as long as the first operand is a string constant.

A file OPEN or CREATE statement will accept a RECS field for compatability with CBASIC. The expression is ignored.

The reserved words LT, GT, GE, LE, EQ, and NE may be used in place of the relational operators <, >, <=, >=, =, and <>.

The following form of an IF statement is supported:

IF <expression> THEN <label>

F. SUBJECT INDEX

A

ABS, 44
ASC, 47
ATN, 44
Arithmetic Operators, 34, 38, 40
Arrays, 20
Assignment Statements, 41

B

B compiler toggle, 110
Binary constants, 7

C

C compiler toggle, 110
CALL Statements, 27
CB80 Character Set, 2
CB80 library, 3
CHAIN Statements, 22
CHR$, 47
CLOSE Statements, 89
COMMAND$, 50
COMMON Statements, 21, 22, 23, 29
CONCHAR%, 77, 79
CONSTAT%, 77
COS, 44
CREATE Statements, 81
Constants, 2, 5, 32
Continuation character, 3, 13, 24

D

D compiler toggle, 111
DATA Statements, 75
DELETE Statements, 89

DIM Statements, 29, 33
DIM statements, 2Ø
Data statements, 9
Data types, 18
Declaration group, 11, 29
Declarations, 18
Dynamic range of numbers, 18

E

ERR, 51
ERRL, 51
EXP, 45
Evaluation of Expressions, 42

F

FLOAT, 45
FRE, 51
File PRINT Statements, 87
File READ Statements, 84, 85
File identification numbers, 81, 84
File name conventions, 9
Fixed files, 8Ø, 85
Formal parameters, 3Ø

G

GET, 91
GOSUB Statements, 3Ø

H

Hexadecimal constants, 7

I

I compiler toggle, 111
IF END Statements, 9Ø
IF Statements, 37, 59
INCLUDE Directive, 16
INKEY, 78
INP, 78
INPUT Statements, 7Ø, 77, 79
INT and INT%, 45
INTEGER Statements, 21
Identifier, 13
Identifier Usage, 24

Identifiers, 2, 4, 27
Identifiers and Reserved Words, 3
Integer constants, 6
Integer numbers, 18

L

L compiler toggle, 111
LEFT$, 47
LEN, 47
LK8Ø, 3
LOCK, 91
LOG, 45
Labels, 24, 29, 53, 6Ø, 63
Licensing Guide, 3
Listing Control Directives
%EJECT, 15
%LIST, 15
%NOLIST, 15
Logical Operators, 34, 35

M

MATCH, 48
MFRE, 51, 82
MID$, 49
MOD, 45
Multiple Line Functions, 26, 53, 6Ø, 61, 66, 1Ø4
Multiple line functions, 12

N

N compiler toggle, 111
NPUT LINE Statements, 69
Numeric Constants, 6
Numeric constant, 12, 13
Numeric constants, 5, 7

O

O compiler toggle, 111
OPEN Statements, 81

P

P compiler toggle, 111
PEEK, 78
POS, 79